ESD-TR-87-149

MTR-10188

AN INTELLIGENT ACQUISITION SUPPORT TOOL

By

RICHARD M. ADLER

MAY 1987

Prepared for

DEPUTY COMMANDER FOR DEVELOPMENT PLANS AND SUPPORT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE

Hanscom Air Force Base, Massachusetts

Project No. 572W

Prepared by

THE MITRE CORPORATION
Bedford, Massachusetts

Contract No. F19628-86-C-0001

ADA183053

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

June I. R. Babson

JUNE I. R. BABSON, Major, USAF
Chief, ESD/MITRE Software Center

FOR THE COMMANDER

Robert J. Kent

ROBERT J. KENT
Director, Software Design Center
Deputy Commander for Development
Plans and Support Systems

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release, distribution unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MTR-10188 ESD-TR-87-149	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION The MITRE Corporation 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Burlington Road Bedford, MA 01730	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Deputy Commander (continued) 8b. OFFICE SYMBOL (If applicable) XRS2	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F19628-86-C-0001
8c. ADDRESS (City, State, and ZIP Code) Electronic Systems Division, AFSC Hanscom AFB, MA 01731-5000	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. 572W / TASK NO. / WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) An Intelligent Acquisition Support Tool	
12. PERSONAL AUTHOR(S) Adler, Richard M.	
13a. TYPE OF REPORT Final 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) 1987 May 15. PAGE COUNT 85
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Artificial Intelligence (AI) Knowledge-Based System Specifications Assistant

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This paper describes SEIMOAR, an AI-based acquisition support tool. SEIMOAR provides a framework for representing and analyzing the functionality, structure, and behavior of C3I systems. Functional and structural information is captured via frame-based templates, in a modeling library that depicts common C3I system capabilities and components. System models are constructed by copying and editing appropriate templates. SEIMOAR also incorporates an object-oriented simulator to support dynamical modeling of system behaviors. An example acquisition task illustrates SEIMOAR's current capabilities. Work in progress on tool extensions is also outlined.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL Diana F. Arimento	22b. TELEPHONE (Include Area Code) (617)271-7454	22c. OFFICE SYMBOL Mail Stop D230

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

8a. for Development Plans and Support Systems.

ACKNOWLEDGMENT

This document has been prepared by The MITRE Corporation under Project No. 572W, Contract No. F19628-86-C-0001. The contract is sponsored by the Electronic Systems Division, Air Force Systems Command, United States Air Force, Hanscom Air Force Base, Massachusetts 01731-5000.

Acknowledgments are also due to MITRE staff who contributed to this effort. Laura Clark implemented the prototype's user interface features, incluiding the menu drivers, the scenario generator, and the iconic graphics and animations. Rich Hilliard, John Maurer, and Mary Lou Urban provided valuable information on acquisition program support in general and on the test vehicle A-specification in particular. Finally, MITRE management, notably Edward L. Lafferty, Judy A. Clapp, Thomas F. Saunders and Myra Jean Prelle helped to define the task scope and to strengthen our presentations of work in progress.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

SECTION 1

INTRODUCTION

Government acquisition programs follow a standardized development cycle, ranging from concept exploration and system definition through to implementation and fielding. The results of early cycle phases are captured in a sequence of progressively more detailed system descriptions, characterizing system functional requirements, architectural specifications, and multiple design iterations.

For the large and complex computer and communications systems commonly needed today, it is becoming increasingly difficult to maintain and analyze the information contained in developmental system descriptions. Analysis encompasses verification of system descriptions with respect to completeness, consistency, technical and economic feasibility. Comparative analysis is also critical, both of variants at a given development phase and of system descriptions across cycle stages. Additional recurring acquisition problems include information accessibility and refinability, and reusability of fragments of existing system descriptions. These problems are not unique to the Government sector: support problems for large system engineering projects are ubiquitous.

A variety of computerized automation tools already exist to support system development activities, including simulation and design, document preparation, and project management. Unfortunately, existing tools, taken individually, are often sharply restricted in functionality and may require significant programming expertise. More seriously, existing tools are extremely difficult to integrate. Simulators and design tools, for example, generate and operate on formal system models or model fragments, whereas document preparation tools manipulate textual descriptions of systems.

This paper describes MITRE's Systems Environment for Intelligent MOdeling and Analysis of Requirements (SEIMOAR), an acquisition support tool based on artificial intelligence (AI) technologies. SEIMOAR addresses the problems of maintenance, analysis capabilities, and tool integration by capturing and manipulating early acquisition cycle system descriptions in the form of symbolic models.

Symbolic models provide explicit representations of system architectural structures, functionality, and behaviors. System design constraints (e.g., performance requirements, maintainability, cost and size relationships, reliability and quality factors), can also be stored as symbolic structures. Symbolic representation and reasoning techniques provide significant advantages over conventional data bases and programming techniques in terms of data compaction, model expressiveness, and general capabilities for manipulating models.

Integrability problems are addressed by requiring all such developmental models to be cast in a uniform representational format. Such models will obviously vary in the level of detail (e.g., overall system capabilities expressed as a collection of functions versus an allocation of functions to specific hardware and software components proposed in a contractor's design), but not in the kinds of symbolic structures used to represent such information. Moreover, all tools required for acquisition support will be constructed on the basis of this common modeling structure. As noted above, automated tools are needed to facilitate generation, maintenance, revision, review, and analysis activities on system development information.

SEIMOAR employs a modeling library approach for the creation of symbolic system descriptions. New systems are very rarely totally unique; typically, they copy or copy and adapt functions and components from existing systems. SEIMOAR exploits this fact by incorporating a modeling library consisting of reusable templates describing generic structural and functional system elements. Templates are represented via frames, which are AI data structures that symbolize objects or object classes in terms of a set of attributes and relations.

In the SEIMOAR prototype, the modeling library describes prototypical functions and components for military Communications, Command, Control, and Intelligence (C^3I) systems. Example system elements templates include signal processing functions and local area networks, characterized by attributes such as security requirements and network protocols. Alternative libraries would be used to support different systems engineering applications domains.

Developmental system models are generated through a simple copy-and-edit strategy: users select and then customize generic library templates, thus representing particular application system model elements. Customization proceeds by supplying values to template-defined attributes, or by defining new attributes and then supplying values. The customized templates are connected together to form a comprehensive model by various relations (e.g., structural connectivity, functional flow, class-subclass and class-instance

links). In AI parlance, system models in SEIMOAR consist of knowledge bases containing structured collections of customized templates.

In addition to representing system architectural structure and functionality, SEIMOAR system models capture system behaviors, the sequences of events involving system components and data objects that implement particular functions. SEIMOAR incorporates a discrete time functional simulator shell for dynamic behavioral modeling of system functionality.

Briefly, users construct behavioral models by providing a specification of flows (e.g., function sequencing and branching), executable procedures associated with individual functions that describe the desired actions on model objects (i.e., system elements and data objects), and iconic animations of these actions. SEIMOAR also incorporates a menu-driven scenario generator to produce sequences of test events (e.g., arriving messages or signal pulses). The simulator uses these ingredients to cycle through applications of functional actions to alter model system element and test data object states over time.

The remainder of this report is divided into three major sections. The first establishes the project context: the portions of the acquisition process to be addressed by SEIMOAR are explained; general problems in the acquisition process are summarized; existing automation tools are analyzed with respect to their inability to resolve current acquisition problems; and finally, a strategy to resolve acquisition process and tooling problems is put forward.

The second section reviews the results of the SEIMOAR project to date. Project logistics and test vehicles are outlined, followed by a description of the AI-based development environment tools used to implement SEIMOAR. The major architectural features of the system prototype are then explained in detail: the modeling library for structures and functions, and the behavioral simulator shell. The explanations incorporate examples from the test vehicle system model and scenarios that suggest SEIMOAR's intended usage.

The third section summarizes limitations in the current version of SEIMOAR, accompanied by design sketches of planned remedies and enhancements. Planned improvements include enhancing the user interface, broadening the representational scope of current library templates, refining the implementation of the base representation model, and introducing automated static and dynamic analysis capabilities.

Appendix A provides a brief tutorial on standard AI knowledge representation techniques. Since sections 3 and 4 rely heavily on the terminology and concepts set forth in the appendix, newcomers to AI are advised to review appendix A before reading the indicated sections. Appendix B describes and compares two prominent commercial AI system building shells.

SECTION 2

BACKGROUND INFORMATION

2.1 DEVELOPMENTAL PHASES OF ACQUISITION PROGRAMS

What are the automation needs in the early phases of systems acquisition? Some initial background information outlining the structure of the procurement process is required in order to frame an answer.

The first major milestone in an acquisition program is the specification of functional requirements for a proposed system. The document that sets out this information is called a System/Segment Specification (SSS). Its ingredients and format are spelled out in the current Department of Defense acquisition process specification, DOD-STD-2167. Figure 1 depicts a tree graph that represents the categories of information (i.e., paragraph headings), of the SSS.

The SSS summarizes the results of a process of system concept definition and exploration. It defines the basic purpose of the system, together with a detailed characterization of desired functional capabilities, qualitative and quantitative performance needs. In addition, the specification delineates requirements for interfacing to related, existing systems, quality and reliability factors, logistics, environmental operating conditions, and other design implementation constraints. Requirements are developed by government customers (e.g., the U.S. Air Force), sometimes in conjunction with a supporting acquisition organization such as MITRE.

The next phase in the process is to select contractors to design and construct the system in question. Source selection is based on a comparative analysis of proposals solicited from potential system developers, performed by an acquisition organization. Proposals outline contractors' projected technical, methodological, and project management approaches to the problems and needs set out in the functional decomposition specifications.

Following source selection, the winning contractor produces a sequence of specification and design products for formal review and analysis by the customer and its acquisition agents. The contractor products describing software aspects of systems up to the implementation phases include: Software Requirements Specifications (SRS), Software Top Level Design Documents (STLDD), and Software Detailed Design Documents (SDDD). Separate interface description

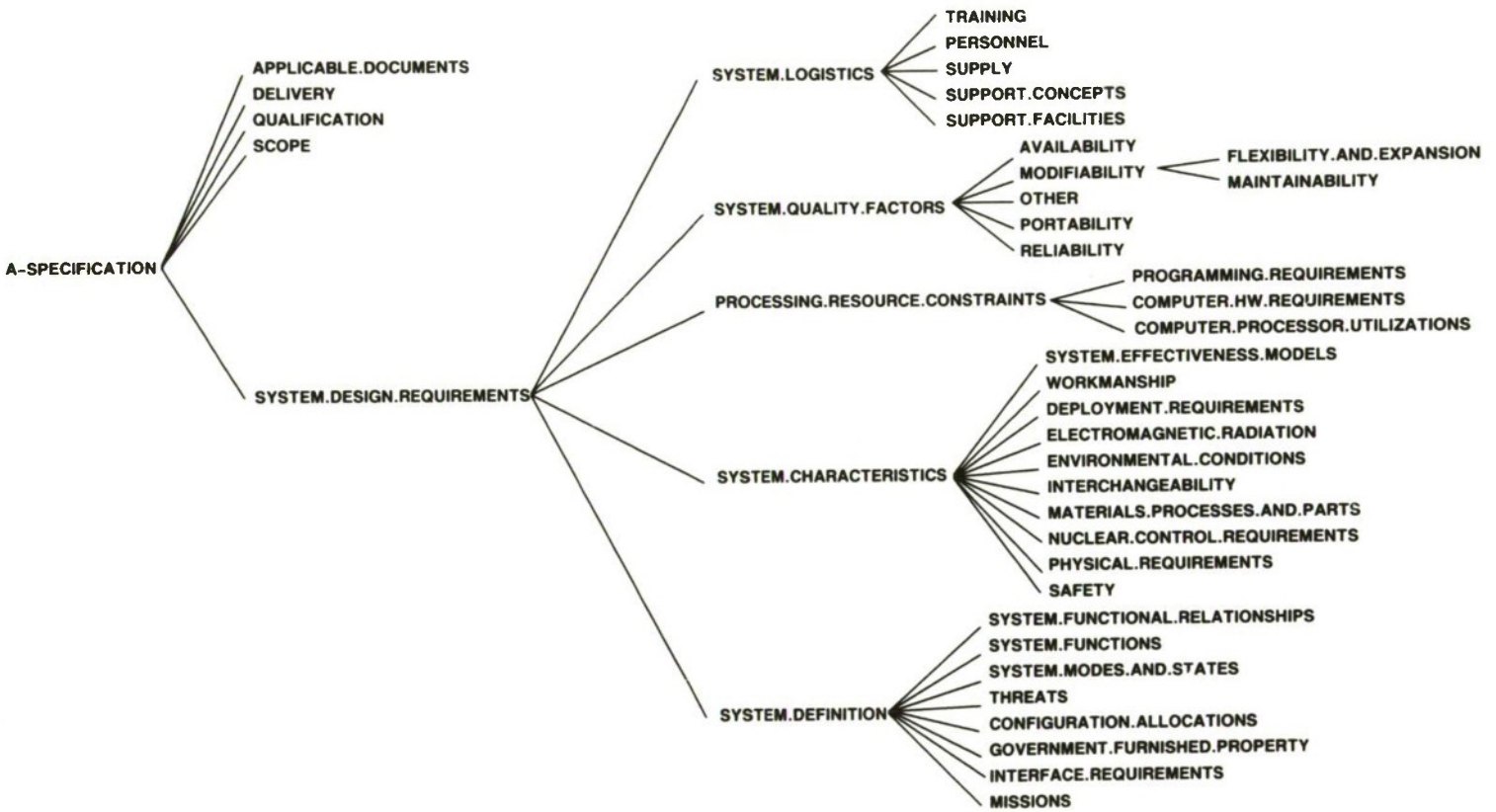

Figure 1. Functional Requirements Information Categories DOD-STD-2167 (SSS)

documents are prepared for each specification or design phase. Each acquisition phase also calls for project management documentation, describing schedules, cost and sizing data, policies, procedures, and methodologies.

To elaborate briefly, the SRS presents a definition of an overall system architecture and configuration allocation: the SRS allocates functions among specific hardware and software subsystems or components, respectively called Hardware and Computer System Configuration Items (HWCIs, CSCIs). The STLDD constitutes a preliminary software specification, which defines data structures and establishes control and data flows between CSCIs and HWCIs. The SDDD characterizes software to a finer-grained level of detail, breaking down CSCIs into modules of pseudocode or a program description language (PDL). In short, the SDDD delineates the internal structure and organization of specific software subsystems.

It should be clear that this kind of iterative specification and design process is not unique to Government organization programs. Large computer system development proceeds similarly in industrial, financial and other sectors as well. Moreover, the structured design cycle sketched here is followed whether systems are contracted externally or constructed in-house. If systems are procured, system design is performed by contractors; otherwise, this role is performed by internal development organizations. Thus while this paper reports on a system development support tool in the context of Government acquisition programs, the results are generalizable to most system development environments.

2.2 RECURRING PROBLEMS IN THE CURRENT ACQUISITION PROCESS

Acquisition support encompasses generation of SSSs, source selection, and review and analysis of subsequent contractor system development products. A variety of problems recur chronically in carrying out these activities within the acquisition process as it is currently structured:

Inaccessibility

It is difficult to selectively retrieve, arrange, and display the contents of system descriptions. In other words, system descriptions are not readily browsed or navigated.

Nonmanipulability

System description information is not in forms suitable for direct experimentation, revision, and refinement. One important

aspect of manipulability is the capability to comprehend or understand systems by examining their (simulated) behavior under controlled stimuli. Another dimension of manipulability is the capability to preserve appropriate structures and relations when system concepts are modified. Both aspects are important in exploring requirements or design variations, and in refining system descriptions.

Unanalyzability

System description information is generally not in forms suitable for direct application of analytical tools.

Nonreusability

System description information is not in forms suitable for being extracted and adapted, piecemeal from existing systems, to be incorporated into specifications and designs of new systems.

2.3 CURRENT AUTOMATION TOOLS AND THEIR LIMITATIONS

A variety of automation tools for acquisition support already exist: document preparation tools, such as word processing and graphics generation systems; program management aids, such as project task decomposition and scheduling tools; behavioral analysis tools, such as simulators; and design tools, such as structured analysis and design systems. Unfortunately, these tools do not address the problems listed above very effectively. In trying to devise a better approach, it is instructive to look closely at the deficiencies of current tooling.

Document preparation systems have serious functional limitations. Word processors basically index and operate on character strings. Keyword searches, for example, depend on exact or near exact character string matches. A keyword search based on "architecture," though, will miss references to "topology," despite the close semantic overlap of the two phrases. Similarly, a search on "computer" will fail to retrieve references to "processor," "workstation," and "mainframe." Moreover, keyword searches will not work at all for information encoded in tables and diagrams.

System developers and reviewers, however, need to extract and organize system descriptions using semantic rather than syntactic references -- concepts, generalized classes and relationships between system elements or element classes. Moreover, given the

importance of information summaries in tables and diagrams, it is critical that graphic as well as textual references be accessible and retrievable.

The mechanical, syntactic character of document processing tools also precludes system description reusability. Document portions can certainly be mechanically copied and ported. However, this rote duplication of form generally requires subsequent manual adaptation of content. For complex CSCIs, manual tracking of the consequences of modifications to components that are closely coupled is difficult and tedious. Reproductory tools based on semantic information about system descriptions would provide superior adaptive capabilities: automated customization tools would perform, or at least enumerate and advise on revisions needed to preserve completeness and consistency.

Current structured analysis and design tools have more reasonable semantic strength: they operate in terms of abstractive classes and relations, such as inputs, outputs, functions, and precedence relations. Unfortunately, such tools generally only capture partial system models for specific system development phases (e.g., design). Such tools usually do not support explicit representation of, nor reasoning about, logistics, quality and reliability factors, and other critical design implementation constraints.

Simulation tools are similarly limited in scope. Moreover, these tools often require programming expertise, making them unsuitable for a general acquisition community. Modeling tools that are not difficult to use are often highly restrictive, narrow purpose systems, such as network performance modelers. This reflects the usual tension in tool design between ease of use as opposed to programmable functionality.

Perhaps the most serious problem with existing tools is that they are not integrated or, for that matter, integrable. Structured analysis and design tools are effective for system design efforts, but not for initial requirements generation, configuration allocation, or traceability between those phases and design. Formal analysis is either performed manually or through piecemeal automated tooling, again operating on partial system models. This fragmentary approach generally precludes comparative analysis either between variant designs or requirements or between system descriptions from different stages of the system development cycle.

The important thing to note in the above critique is that individual tools are deficient in different respects. In particular, word processors, most project management tools, and graphics

preparations systems have serious functional inadequacies, whereas design and modeling tools suffer primarily from restrictive scope and integrability problems. The following explanation for this distinction provides the primary motivation for the SEIMOAR project.

The functional capabilities of all computer-based tools derive from their mechanical manipulations of syntactic (data) structures. The critical difference between current document processors and simulators, structured design systems and knowledge-based tools lies in their internal interpretations of the objects that they manipulate.

Word processor operations on character strings model manipulations of words. (Similarly, graphics program operations on icons model manipulations of pictures.) Thus, the immediate interpretation of operations is in terms of their effect on language (pictorial) elements; the additional interpretation of words to a particular domain (e.g., systems acquisition), is totally incidental and outside the scope of the semantics of the tooling. Tool users must explicitly impose this interpretation, and direct tool operations manually (e.g., keyword searches on synonymous phrases).

In contrast, knowledge-based systems, simulators, and design tools operate on symbolic tokens that represent system elements, features or relations among elements. Consequently, an immediate interpretational correspondence exists between tool activities and domain activities (e.g., behavioral simulations, design manipulations). Tool constraints and capabilities directly reflect the semantics of system model structures and manipulations.

The tokens that knowledge-based systems, simulators, and design tools operate on are not words in acquisition documentation, but formal symbolic representations of system models or model fragments. This explains the integrability limitations of such tools with respect to word processors and graphics systems; the latter tools process raw text and diagrams rather than symbolic models.

2.4 SEIMOAR'S STRATEGY -- MODEL-BASED ACQUISITION SUPPORT

The problems in the current acquisition process and tooling can now be seen to be reflections or manifestations of an underlying flaw in the procurement process. The products that drive the current acquisition process are textual (and graphic) descriptions of system models. These are the items that are generated and transmitted between acquisition customers (or agents) and the contractors who develop and implement systems.

Documentation, of itself, is an unsuitable vehicle for system development activities; it records only the results of specification and design processes. Documentation is not the actual product of concept definition, requirements specification or design development. System models, or model fragments are. Accordingly, specification and design analysis and review need to be performed on these products directly. In short, concrete symbolic system models should be the objects that are generated by, manipulated by, and transmitted between acquisition customers and contractors.

Given this diagnosis, it follows that effective automation for acquisition support requires a model-based approach. This strategy depends on two sets of ingredients: a sequence of symbolic system description models, cast in a uniform or canonical representational framework, and an integrated tool set for generating, exploring, refining, analyzing, and comparing such models.

Clearly, the two sets of elements are closely coupled: the form and content of the tooling presupposes a specific representational framework, while actual system models cannot be constructed and manipulated without the tooling. This coupling seems to be characteristic of integrated environment architectures: it is necessary to determine both the kinds of information that must be represented, simpliciter, and the kinds of activities and operations that are going to be performed on that information.

How, exactly, does this model-based approach address the acquisition support issues described earlier?

Accessibility

Requirements, specifications, and design descriptions are cast as models (actually individual knowledge bases), which can be browsed or explored by abstractive reference and retrieval. System elements are now represented by model entities rather than uninterpreted tokens (e.g., character strings, icons).

Moreover, referencing and manipulation of system elements and relations in symbolic models can now be accomplished in terms of pointing and other operations on graphic icons. In conventional development environments, icons (e.g., Macintosh graphics), are token bitmaps; on par with simple character strings, they lack any semantic (abstractive center) within the tool context. In contrast, model-based representations can explicitly associate icons with particular symbolic model elements (e.g., components or functions). These links support internal semantic interpretations, whereby

operations on iconic tokens (e.g., pointing, connecting) are translated into model element manipulations (e.g., referencing, structural linkage).

Manipulability

Behavioral portions of symbolic system models can be used to drive simulations of system performance, both qualitative and quantitative, based on controlled test stimuli. Equally important, explicit constraint relationships imposed on symbolic model elements guarantee maintenance of consistency when model features are permuted or perturbed in exploring system variants (e.g., alternative system architectures or functional requirements).

Analyzability

System descriptions, now cast as symbolic models within a canonical representational framework, can be operated on by a uniform set of analytical tools. This is the value of integrability -- one model, cast in a single representational format, is accessible to any tool that presupposes that format. Moreover, if analytical results preserve that format, such output can be manipulated by further tools (e.g., as in UNIX pipes).

Reusability

System description fragments, now cast as pieces of symbolic models can be extracted and adapted using model manipulation tools. Specifically, model elements describing generic classes of functions or components can be specialized to subclasses suitable to given system applications.

SEIMOAR embodies this model-based approach to system development. It provides a uniform representational framework for expressing symbolic models of system structural, functional, behavioral and contextual (i.e., design constraint) information. SEIMOAR will also incorporate an integrated tool set, based on this framework, for generating, manipulating and analyzing symbolic system models.

Tools are being designed to perform completeness, consistency, and technical feasibility (e.g., performance verification) analyses. Symbolic model system descriptions will be assessed both individually and comparatively (i.e., across models representing different development stages). Other planned utilities include sizing and costing estimation tools for system descriptions at functional

requirements and specification phases, and tools for exploring model variants at a given developmental stage (e.g., alternative architectures, functional decompositions).

In short, SEIMOAR is intended to constitute a strongly automated acquisition support environment that will facilitate requirements definition and assessment of contractor (or in-house developer) deliverables. The phases of system development that SEIMOAR addresses are specification through early (high-level) design.

SECTION 3

SEIMOAR

3.1 SEIMOAR PROJECT HISTORY

The initial version of SEIMOAR was implemented using KEE (Version 2.1, Intellicorp), a commercial AI system-building tool hosted on a Symbolics LISP minicomputer. The choice of environment was straightforward, given the decision to adopt a symbolic model-based approach to acquisition support: LISP machines provide the best available combination of development utilities, memory capacity and symbolic processing power.

Time and staffing constraints dictated the use of a commercial tool shell for system prototype development. MITRE already possessed a license for KEE, thus determining a specific choice of commercial development environment. Project resources did not cover purchase of SIMKIT, a companion product to KEE that provides discrete event simulation capabilities. Consequently, a custom-built dynamic simulator was constructed and incorporated into SEIMOAR.

To date, the SEIMOAR project has expended roughly eight staff-months of technical effort: the two project staff members learned how to use KEE, designed and implemented the SEIMOAR tool, inclusive of the dynamic simulator shell, and constructed a functional requirements model for a test application system. The author designed and implemented most of the SEIMOAR shell and the test application system model. A junior associate working half-time on the project implemented SEIMOAR's graphics and current user interface.

3.2 SEIMOAR TEST VEHICLES

A functional requirements specification for a modest-sized (100K SLOC) military communications system (MCS) was selected as an application test vehicle. The goal of the test vehicle program is to enhance local communications capabilities for an existing distributed Command and Control System.

Each MCS site is to consist of multiple workstations, coupled via a local area network to a central processor hosting an automated message handling system (AMH). The AMH, the core of MCS test vehicle, automatically logs and distributes messages to site users.

Message review, drafting, and sending capabilities are also specified for the AMH. Additional MCS functionality pertains to maintainability, workstation capabilities, AMH operations support, security, statistics, and testing.

SEIMOAR was developed by analyzing the MCS functional requirements. Representational needs were determined based on the kinds of information found in the specification and the kinds of anticipated activities (e.g., generation, review and refinement, analysis) involving that data. This constitutes a reversal of the tool's intended mode of operation, which is to construct and manipulate system models using the tool's capabilities. Preexisting functional requirements were employed to avoid having to combine initial tooling design and implementation work along with construction of a new system specification.

In the development process, valuable insight was gained into the adequacy of the MCS functional requirements themselves. Several important errors and ambiguities were uncovered, the most important of which are reviewed in section 3.7. This exercise thus helped to substantiate the claimed value of SEIMOAR as a system development environment: the representational framework enforces a uniform methodology that highlights ambiguities, omissions, and inconsistencies in evolving system descriptions.

The MCS functional requirements are unusually detailed for an initial system definition. In addition to a characterization of desired functionality, a high-level system architecture and a partial configuration allocation are stipulated. Typically, such information is provided by the contractor rather than by the customer, and somewhat later in the acquisition cycle. SEIMOAR's capability to encode such data helps to validate the tool's design, despite the initial, restricted development basis of a single acquisition product.

It is anticipated (cf section 4) that the representational apparatus assembled to capture and manipulate functional requirements system information will have to be extended somewhat to model subsequent system descriptions in the acquisition cycle. Project plans call for exercising the tool against functional requirements, specifications, and early design phases of further system acquisitions. This strategy will serve to define requisite extensions and to provide substantive confirmation of the basic adequacy of SEIMOAR's modeling approach and representational framework.

3.3 KEE DEVELOPMENT ENVIRONMENT

Knowledge Engineering Environment (KEE, Intellicorp) is a hybrid tool for building AI systems. Hybrid shells incorporate elements extracted from several distinct knowledge representation models or paradigms within a single, integrated framework. KEE relies on a basic frames language that has been augmented to support capabilities borrowed from object-oriented and rule-based technologies as well. The following description pertains to KEE Version 2.1. A supplemental explanation of standard artificial intelligence representation techniques (e.g., frames), is provided in appendix A of this report. Appendix B compares KEE with Automated Reasoning Tool (ART, Inference Corp.), a competitor commercial shell.

KEE frames, called units, have the normal internal structures, slots and facets. Frames can be linked by class-subclass and class-member relations. KEE's inheritance mechanisms are unusually rich. Slots are categorized into two types, OWN and MEMBER, corresponding to class and instance attributes, respectively. Instance slots (not slot values) are inherited from class to subclass or member frames, but class slots are not. Thus, a descriptor appropriate only to classes (e.g., GREATEST-AGE, CLASS-POPULATION) need not be inherited by subclass or member frames from class units.

KEE allows for multiple inheritance of slots. This means that a class can inherit MEMBER slots from multiple superclasses. Zetalisp Flavors provides an analogous capability (i.e., metaclass mixins). This capability is extremely useful in cases where generic attributes are naturally partitioned among different superclasses. For example, graphics attributes (bitmaps, mouse behaviors) are naturally grouped according to an image object class, while domain model constituent attributes are best organized about structural entity classes. Multiple inheritance enables selective combination of both kinds of properties in a single class, or separate property class ascription, as necessary.

Propagation of VALUES facet data from class to subclass and member units is regulated by an INHERITANCE facet. KEE provides about a dozen standard inheritance relations and supports user programmed ones as well. For example, OVERRIDE.VALUES specifies transmission of class values to subclasses or members as defaults, which can be overridden by explicit subclass or member data (e.g., GREEN is the default VALUES datum for COLOR for all instances of GRANNY.SMITH.APPLES, unless explicitly overridden. UNIQUE blocks transmission of superclass values, while UNION combines superclass values with any explicitly supplied data for the inheriting frame.

KEE provides a special (Boolean) language for the VALUECLASS facet, which flags admissible or legal data classes for the VALUES facet. For example (ONE.OF arg1 arg2 ...) expresses the restriction that candidate data for the VALUES facet must match the member arguments. Two cardinality slots allow specification of maximum and minimum allowable number of data items for the VALUES facet, while a COMMENT facet allows slot-level documentation. User-defined facets are also available.

KEE frames support two kinds of procedural attachments, methods and active values. Methods are implemented at the slot level, as LISP code stored in the method's VALUES facet. Method slots have a special INHERITANCE facet mode to support transmission of code between frames along class-subclass and class-member links. Methods are activated by using a KEE function that sends a message consisting of the method slot name and appropriate arguments to the relevant frame. This model of communication and code execution is borrowed from object-oriented knowledge representation models and programming languages.

Frame-based procedural attachments, called active values or demons, attach to frame slots. As expected, KEE implements demons at the facet level. Predefined facets allow the definition of LISP functions that are activated, as needed, when slot VALUES data items are added, removed, or retrieved.

KEE implements rules (and rule classes) via frames. A rule frame has slots that contain a first-order predicate calculus representation of rule premises (if clauses) and conclusions (then clauses). The contents of these slots are operated upon by KEE's built-in rule inference engines. Both backward and forward chaining mechanisms are provided. Invocation of either rule engine requires the specification of a particular rule class (i.e., as the functional argument). Rule classes constitute KEE's mechanism for partitioning large rule bases into manageable subsets that can be examined selectively.

The rule calculus itself is not particularly intuitive, but KEE users actually read, write, and edit rules using an "external-form" slot, which employs a simple English-like rule language. The rule engines can also be invoked interactively, for knowledge base queries and manipulations, using a special declarative language. KEE contains utility functions for converting between frame, predicate calculus, and the declarative language representations.

In addition to these basic representation and reasoning capabilities, KEE supplies various system development environment

facilities: a strong user interface and tools for building custom application interfaces (e.g., menus, windows, mouse pointers, bitmap graphics editor), incremental compilation, debugging and monitoring capabilities such as rule traces and breakpoints, and language-based editors. Moreover, KEE utility functions for manipulating knowledge bases are available to applications programmers. Another helpful utility is KEE's graphic tree browser, which depicts the units and their class relations for the specified knowledge base.

KEE graphics are also implemented through the frames system. Image classes are created that point to icons generated through a bitmap graphics editor. Instance frames can then be created, associating icon instances with particular windows and window coordinates. KEE also allows for composite icon image classes and instances, called panels. Panels constitute the equivalent of a graphic macro, by combining multiple icons in specific, fixed relations (spatial and computational) to one another.

Finally, KEE provides a special class of icons, known as active images, that constitute a graphic variant of demons. Active images are icons that attach to frame slots. Their coupling is dynamic: modifications to slot VALUES data items are reflected in icon appearance. For example, changes in a thermometer reading change a corresponding object's temperature slot datum. This capability is extremely useful for graphic depictions of simulations. A subclass of active images, called actuators provide two-way coupling: actuator icon changes (i.e., from mouse actions), are reflected in frame slot data changes, as well as the converse.

3.4 LIBRARY-BASED SYSTEM MODELING IN SEIMOAR

A central methodological tenet of the SEIMOAR project was that system description generation and analysis activities should be driven by modeling libraries.

MITRE's primary expertise is in acquisition of C^3I systems. These systems constitute a relatively well-defined class: new designs generally consist of variations on a small number of standard subsystems and components (e.g., sensors, platforms, and processors), coupled together by one of a relatively few kinds of architectural configurations (e.g., network topologies or bus structures). Thus, new systems typically copy or copy and adapt fairly generic bands of functions, components, and interconnections. Consequently, the application domain is inherently well-suited to description and design methods employing templates.

Templates, in this context, are simply prototypical system model elements, characterized in terms of idealized, generic properties and relations. Examples include physical characteristics of model elements (e.g., size, weight), internal structures, (e.g., components connected together), and operational descriptors (e.g., purpose, capacity, processing speed).

SEIMOAR's current modeling library represents C^3I functions, systems, hardware and software components using templates implemented in terms of KEE frame classes (unit classes). For other application domains amenable to the library-based development approach (e.g., weapons or commercial data processing systems), appropriately different frame template collections would be substituted in place of SEIMOAR's present library.

System models in SEIMOAR are comprised of knowledge bases containing collections of frames. System model generation using SEIMOAR proceeds according to a simple copy-and-edit strategy: generic library templates are copied into a system model knowledge base and customized by incorporating characteristics specific to capabilities or structures desired in the new system.

3.4.1 Template Customization -- General Scenarios and Examples

Template customization can take place in two ways. First, templates might simply be filled in by supplying data for slot VALUES facets. A second form of customization is to modify the template itself by adding slots, and then supplying data items to fill the new slots' VALUES facets.

An example of model element generation from the application test vehicle is illustrated below. Figure 2 displays slots representing some of the attributes that characterize communications interfaces as a general class. Note that the KEE unit header contains class and knowledge base pointers, and that some slots are created by, and maintained for KEE itself. TRANSMISSION.CODE illustrates a value class restriction, indicating that either or both BLOCK and CONTINUOUS are admissible data for the VALUES facet for this descriptive attribute.

Figure 3 displays a more specialized interface template, which adds attributes that characterize hardware and software requirements for the seven-layer ISO model of network protocols. Many, though

Unit: **COMMUNICATIONS.INTERFACES** in knowledge base **MODEL.LIBRARY**
Created by Adler on 3-Jun-86 13:47:57
Modified by jam on 10-Oct-86 11:15:39
Superclasses: **HARDWARE.OBJECTS**
Subclasses: **ISO.MODEL.COMMUNICATIONS.INTERFACES**
Member Of: **(CLASSES** in kb **GENERICUNITS)**

Member slot: **DATA.TRANSMISSION.MODE** from **COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
ValueClass: (ONE.OF BLOCK CONTINUOUS)
Comment: "modes supported"
Values: Unknown

Member slot: **DESCRIPTION** from **MODEL.ELEMENTS**
Inheritance: **UNIQUE.VALUES**
Comment: "descriptive text describing object"
Values: Unknown

Member slot: **FUNCTIONALITY** from **COMMUNICATIONS.INTERFACES**
Inheritance: **UNIQUE.VALUES**
Comment: "pointer to associated functions"
Values: Unknown

Member slot: **HW.REQUIREMENTS** from **COMMUNICATIONS.INTERFACES**
Inheritance: **UNIQUE. VALUES**
Comment: "explicit hw requirements for interface"
Values: Unknown

Member slot: **IMPLEMENTATION.REQUIREMENTS** from **COMMUNICATIONS.INTERFACES**
Inheritance: **UNIQUE. VALUES**
Comment: "generalized constraints on interface design"
Values: Unknown

Member slot: **INTERFACED.OBJECTS** from **COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Comment: "objects between which interface is defined"
Values: Unknown

Member slot: **PURPOSE** from **MODEL.ELEMENTS**
Inheritance: **UNIQUE.VALUES**
Comment: "description of the purpose the given object is supposed to satisfy or contribute to"
Values: Unknown

Member slot: **TRANSMISSION.CHANNEL.TYPE** from **COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
ValueClass: (ONE.OF FULL HALF)
Values: Unknown

Member slot: **TRANSMISSION.CHANNEL.TYPE** from **COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
ValueClass: (ONE.OF FULL HALF)
Values: Unknown

Member slot: **TRANSMISSION.CODE** from **COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
ValueClass: (ONE.OF ASCII BCD)
Values: Unknown

Member slot: **TRANSMISSION.MODE** from **COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
ValueClass: (ONE.OF DIGITAL ANALOG)
Values: Unknown

Member slot: **TRANSMISSION.TYPE** from **COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
ValueClass: (ONE.OF 300 1200 2400 4800 9600 19200 56000)
Comment: "list of line speeds (in bits per second) supported by comm. channel"
Values: Unknown

Member slot: **TRANSMISSION.RATE** from **COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
ValueClass: (ONE.OF SYNCHRONOUS ASYNCHRONOUS)
Values: Unknown

Figure 2. Communications Interfaces Library Template

Unit: **ISO.MODEL.COMMUNICATIONS.INTERFACES** in knowledge base **MODEL.LIBRARY**
Created by Adler on 3-Jun-86 13:49:09
Modified by Adler on 3-Jun-86 14:01:59
Superclasses: **COMMUNICATIONS.INTERFACES**
Subclasses: (**CU.INTERFACES** in kb **WISCUS**)
Member Of: (**CLASSES** in kb **GENERICUNITS**)

Member slot: **APPLICATION.LAYER** from **ISO.MODEL.COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Comment: "seventh lowest (top) of ISO seven-layer network model. Services are provided that directly support user and application tasks and overall system management, such as resource sharing, file transfers, remote file access, data base management, and network management."
Values: Unknown

Member slot: **DATA.LINK.LAYER** from **ISO.MODEL.COMMUNICATIONS.INTERFACES**
Inheritance: OVERRIDE.VALUES
Comment: "Second lowest of ISO seven-layer network model. Establishes an error-free communications path between network nodes over the physical channel, frames messages for transmission, checks integrity of received messages, manages access to and use of the channel, ensures proper sequence of transmitted data."
Values: Unknown

Member slot: **INTERFACED.OBJECTS** from **COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Comment: "objects between which interface is defined"
Values: Unknown

Member slot: **NETWORK.CONTROL.LAYER** from **ISO.MODEL.COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Comment: "third lowest of ISO seven-layer network model. Addresses messages, sets up the path between communicating nodes, routes messages across intervening nodes to their destination, and controls the flow of messages between nodes."
Values: Unknown

Member slot: **PHYSICAL.LINK.LAYER** from **ISO.MODEL.COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Comment: "Lowest level of ISO seven-layer network model. Defines the electrical and mechanical aspects of interfacing to a physical medium for transmitting data, as well as setting up, maintaining, and disconnecting physical links. When implemented, this layer includes the software device driver for each communications device plus the hardware itself - interface devices, modems, and communications lines."
Values: Unknown

Member slot: **PRESENTATION.LAYER** from **ISO.MODEL.COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Comment: "sixth lowest of ISO seven-layer network model. Encoded data that has been transmitted is translated and converted into formats which enable display on terminal screens"

*APPLICATION.LAYER
*DATA.LINK.LAYER
*DATA.TRANSMISSION.MODE
*DESCRIPTION
*FUNCTIONALITY
*HW.REQUIREMENTS
*IMPLEMENTATION.REQUIREMENTS
*INTERFACED.OBJECTS
*NETWORK.CONTROL.LAYER
*PHYSICAL.LINK.LAYER
*PRESENTATION.LAYER
*PURPOSE
*SESSION.LAYER
*TRANSMISSION.CHANNEL.TYPE
*TRANSMISSION.CODE
*TRANSMISSION.MODE
*TRANSMISSION.RATE
*TRANSMISSION.TYPE
*TRANSPORT.LAYER

TEMPLATE ATTRIBUTE LIST

Figure 3. ISO Model Communications Interfaces Template (Partial)

not all communications interfaces can be characterized by supplying values to such a template. The ISO.MODEL.COMMUNICATIONS.INTERFACES template inherits the attributes of its superclass, COMMUNICATIONS. INTERFACES and adds the seven protocol layers as properties.

Depending on the application system, one of three courses of action might be followed in creating a communications interface. If the ISO template were adequate, an acquisition support user would copy the template into the system description knowledge base, rename the unit, create instances of the unit, and fill in appropriate data. Class instances are created following template copy and adaptation because multiple individuals (e.g., LAN interface units) appear both in single systems and multiple copies of systems (e.g., computer networks).

Alternatively, the ISO model might have to be refined into further specialized subclasses (e.g., DECNET interfaces), which fill in the ISO model attribute VALUES facets with specific data. The resulting template, DECNET.INTERFACES, would still be sufficiently general to be broadly useful. New templates, once validated by SEIMOAR support personnel, would be added to the modeling library. The copy and edit procedure described above would then be followed for constructing the system model description.

Third, it might be the case that no suitable template is available (e.g., if the ISO model were unsuitable). In this case, the user would develop a new template subclass of COMMUNICATIONS. INTERFACES and add it to the library, following the template verification procedure. The copy and edit procedure would then be followed once again.

In both of the last two scenarios, the modeling library accumulates new templates over time as users model new systems and components or functions that can be generalized to serve across C^3I acquisitions. It is important to note that SEIMOAR, like most library systems, accommodates change primarily through addition. Restructuring of existing library units is ill-advised: changing the composition of the source templates upon which earlier system models depend after the fact destroys the baseline that drives analysis utilities (section 4.1.6).

Figures 4a, 4b, and 4c reveal how templates are customized in order to describe components in the test vehicle system, the MCS message-handling system. Individual MCS sites are connected to the AUTODIN network via message exchange systems, called AMPEs. There are four different subclasses of AMPE exchanges. Information describing interfaces between MCS sites and particular subclasses of AMPE exchanges (e.g., LDMX), is localized to that subclass frame.

The AMPE INTERFACES Unit in MCS Knowledge Base

Unit: **AMPE.INTERFACES** in knowledge base **MCS**
Created by Adler on 3-Jun-86 15:35:48
Modified by Adler on 14-Oct-88 12:06:20
Superclasses: **AUTODIN.INTERFACES**
Subclasses: **AMME.INTERFACES**, **AFAMPE.INTERFACES.PCTCS.INTERFACES**, **LDMX.INTERFACES**
Member Of: (**CLASSES** in kb **GENERICUNITS**)

Member slot: **APPLICATION.LAYER** from **ISO.MODEL.COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Comment: "seventh lowest (top) of ISO seven-layer network model. Services are provided that directly support user and application tasks and overall system management such as resource sharing, file transfers, remote file access, data base management, and network management."
Values: Unknown

Member slot: **DATA.LINK.LAYER** from **ISO.MODEL.COMMUNICATIONS.INTERFACES**
Inheritance: OVERRIDE.VALUES
Comment: "Second lowest of ISO seven-layer network model. Establishes an error-free communications path between network nodes over the physical channel, frames messages for transmission, checks integrity of received messages, manages access to and use of the channel, ensures proper sequence of transmitted data."
Values: Unknown

Member slot: **DATA.TRANSMISSION.MODE** from **AMPE.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
ValueClass: (ONE OF BLOCK CONTINUOUS)
Comment: "modes supported"
Values: BLOCK CONTINUOUS

Member slot: **DESCRIPTION** from **AMPE.INTERFACES**
Inheritance: **UNIQUE.VALUES**
Comment: "descriptive text describing object"
Values: "class of interfaces to class of AMPE message exchange centers"

Member slot: **FUNCTIONALITY** from **AUTODIN.INTERFACES**
Inheritance: **UNIQUE.VALUES**
Comment: "pointer to associated functions"
Values: Unknown

Member slot: **HW.REQUIREMENTS** from **AMPE.INTERFACES**
Inheritance: **UNIQUE.VALUES**
Documentation: "see IAW FED STDS 1008 1005 1006"
Comment: "possibly required for remote access to AMPE devices- to be provided as part of interfacing hardware"
Values: MODEMS

Member slot: **IMPLEMENTATION.REQUIREMENTS** from **AMPE.INTERFACES**
Inheritance: **UNIQUE.VALUES**
Comment: "switchability constraint required to insure single interface to cu application requirement (cf impl req of autodin interfaces)"

(Output) The ISO.MODEL.COMMUNICATIONS.INTERFACES Unit in MODEL.LIBRARY Knowledge Base

Unit: **ISO.MODEL.COMMUNICATIONS.INTERFACES** in knowledge base **MODEL.LIBRARY**
Created by Adler on 3-Jun-86 13:49:09
Modified by Adler on 3-Jun-86 14:01:59
Superclasses: **COMMUNICATIONS.INTERFACES**
Subclasses: (**CU.INTERFACES** in kb **MCS**)
Member Of: (**CLASSES** in kb **GENERICUNITS**)

Member slot: **APPLICATION.LAYER** from **ISO.MODEL.COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Comment: "seventh lowest (top) of ISO seven-layer network model. Services are provided that directly support user and application tasks and overall system management such as resource sharing, file transfers, remote file access, data base management, and network management."
Values: Unknown

Member slot: **DATA.LINK.LAYER** from **ISO.MODEL.COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Comment: "Second lowest of ISO seven-layer network model. Establishes an error-free communications path between network nodes over the physical channel, frames messages for transmission, checks integrity of received messages, manages access to an use of the channel, ensures proper sequence of transmitted data."
Values: Unknown

Member slot: **DATA.TRANSMISSION.MODE** from **COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
ValueClass: (ONE OF BLOCK CONTINUOUS)
Comment: "modes supported"
Values: Unknown

Member slot: **DESCRIPTION** from **MODEL.ELEMENTS**
Inheritance: **UNIQUE.VALUES**
Comment: "descriptive text describing object"
Values: Unknown

Member slot: **FUNCTIONALITY** from **COMMUNICATIONS.INTERFACES**
Inheritance: **UNIQUE.VALUES**
Comment: "pointer to associated functions"
Values: Unknown

Member slot: **HW.REQUIREMENTS** from **COMMUNICATIONS.INTERFACES**
Inheritance: **UNIQUE.VALUES**
Comment: "explicit hw requirements for interface"
Values: Unknown

Member slot: **IMPLEMENTATION.REQUIREMENTS** from **COMMUNICATIONS.INTERFACES**
Inheritance: **UNIQUE.VALUES**
Comment: "generalized constraints on interface design"
Values: Unknown

Member slot: **INTERFACED.OBJECTS** from **COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Comment: "objects between which interface is defined"

Figure 4a. MCS AMPE Interfaces Unit

The AMPE INTERFACES Unit in MCS Knowledge Base

Member slot: **HW.REQUIREMENTS** from **AMPE.INTERFACES**
Inheritance: **UNIQUE.VALUES**
Documentation: "see IAW FED STDS 1008 1005 1006"
Comment: "possibly required for remote access to AMPE devices- to be provided as part of interfacing hardware"
Values: MODEMS

Member slot: **IMPLEMENTATION.REQUIREMENTS** from **AMPE.INTERFACES**
Inheritance: **UNIQUE.VALUES**
Comment: "switchability constraint required to insure single interface to cu application requirement (cf impl req of autodin interfaces)"
Values: SHOULD PRESENT SINGLE INTERFACE TO AMPE SYSTEMS
SHOULD NOT FORCE MODIFICATIONS TO AMPE SYSTEMS
SHALL SUPPORT SITE UNIQUE IMPLEMENTATION REQUIREMENTS
SHALL SUPPORT SWITCHING AMONG AMPE INTERFACE TYPES AT ANY CONFIGURATION SITE

Member slot: **INTERFACE.TYPE** from **AMPE.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Documentation: "see latest revision of functional specification- Army AMPE Interface to Remote Terminals U.S. Army Communications Electronics Engineering Installation Agency (USACEEIA)"
Values: MART AMPE

Member slot: **INTERFACED.OBJECTS** from **AUTODIN.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Comment: "objects between which interface is defined"
Values: MCS AUTODIN

Member slot: **MESSAGE.TYPES** from **AMPE.INTERFACES**
Inheritance: **UNION**
Comment: "list of ordered triplets in format (message type message format (activities))"
Documentation: "see AUTODIN Operating Procedures (JANAP 128 (I))"
Values: (NARRATIVE GENSER JANAP 128 (RECEIVE TRANSMIT))

Member slot: **NETWORK.CONTROL.LAYER** from **ISO.MODEL.COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Comment: "third lowest of ISO seven-layer network model. Addresses messages, sets up the path between communicating nodes, routes messages across intervening nodes to their destination, and controls the flow of messages between nodes."
Values: Unknown

Member slot: **PHYSICAL.LINK.LAYER** from **ISO.MODEL.COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Comment: "Lowest level of ISO seven-layer network model. Defines the electrical and mechanical aspects of interfacing to a physical medium for transmitting data, as well as setting up, maintaining, and disconnecting physical links. When implemented, this layer includes the software device for each communications device plus the hardware itself - interface devices, modems, and communications lines."

(Output) The ISO.MODEL.COMMUNICATIONS.INTERFACES Unit in MODEL.LIBRARY Knowledge Base

Member slot: **HW.REQUIREMENTS** from **COMMUNICATIONS.INTERFACES**
Inheritance: **UNIQUE.VALUES**
Comment: "explicit hw requirements for interface"
Values: Unknown

Member slot: **IMPLEMENTATION.REQUIREMENTS** from **COMMUNICATIONS.INTERFACES**
Inheritance: **UNIQUE.VALUES**
Comment: "generalized constraints on interface design"
Values: Unknown

Member slot: **INTERFACED.OBJECTS** from **COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Comment: "objects between which interface is defined"
Values: Unknown

Member slot: **NETWORK.CONTROL.LAYER** from **ISO.MODEL.COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Comment: "third lowest of ISO seven-layer network model. Addresses messages, sets up the path between communicating nodes, routes messages across intervening nodes to their destination, and controls the flow of messages between nodes."
Values: Unknown

Member slot: **PHYSICAL.LINK.LAYER** from **ISO.MODEL.COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Comment: "lowest level of ISO seven-layer network model. Defines the electrical and mechanical aspects of interfacing to a physical medium for transmitting data, as well as setting up, maintaining, and disconnecting physical links. When implemented, this layer includes the software device driver for each communications device plus the hardware itself - interface devices, modems, and communications lines."
Values: Unknown

Member slot: **PRESENTATION.LAYER** from **ISO.MODEL.COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Comment: "sixth lowest of ISO seven-layer network model. Encoded data that has been transmitted is translated and converted into formats which enable display on terminal screens and printers - forms that can be understood and directly manipulated by users."
Values: Unknown

Member slot: **PURPOSE** from **MODEL.ELEMENTS**
Inheritance: **UNIQUE.VALUES**
Comment: "description of the purpose the given object is supposed to satisfy or contribute to"
Values: Unknown

Member slot: **SESSION.LAYER** from **ISO.MODEL.COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Comment: "fifth lowest of ISO seven-layer network model. Establishes and controls SYSTEM-DEPENDENT aspects of communications sessions between SPECIFIC nodes in the network and bridges the gap between the services provided by the Transport layer and the logical functions running under the operating system in a participating node."
Values: Unknown

Figure 4b. MCS AMPE Interfaces Unit (2)

The AMPE.INTERFACES Unit in MCS Knowledge Base

Member slot **PURPOSE** from **AMPE.INTERFACES**
Inheritance: **UNIQUE.VALUES**
Comment: "description of the purpose the given object is supposed to satisfy or contribute to"
Values: INTERFACE TO AMPE EXCHANGES ON AUTODIN NETWORKS

Member slot **SESSION.LAYER** from **ISO.MODEL.COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Comment: "fifth lowest of ISO seven-layer network model. Establishes and controls **SYSTEM-DEPENDENT** aspects of communications sessions between **SPECIFIC** nodes in the network and bridges the gap between the services provided by the Transport layer and the logical functions running under the operating system in a participating node."
Values: Unknown

Member slot **TRANSMISSION.CHANNEL.TYPE** from **AMPE.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
ValueClass: (ONE OF FULL HALF)
Comment: "duplex modes- full and -or half"
Values: FULL HALF

Member slot **TRANSMISSION.CODE** from **AMPE.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
ValueClass: (ONE OF ASCII BCD)
Values: ASCII

Member slot **TRANSMISSION.CODE** from **AMPE.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
ValueClass: (ONE OF DIGITAL ANALOG)
Values: DIGITAL

Member slot **TRANSMISSION.RATE** from **AMPE.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
ValueClass: (ONE OF 300 1200 2400 4800 9600 19200 56000)
Documentation: "see IAW joint FIPS PUBS//FED STDS 37//1001. 16-1//1010. 17-1//1011. 22-11013."
Comment: "list of line speeds (in bits per second) supported by comm. channel"
Values: 1200 2400 4800

Member slot **TRANSMISSION.TYPE** from **AMPE.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
ValueClass: (ONE OF SYNCHRONOUS ASYNCHRONOUS)
Values: SYNCHRONOUS

Member slot **TRANSPORT.LAYER** from **ISO.MODEL.COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Comment: "fourth lowest of ISO seven-layer network model. Provides end-to-end control of a communication session once the path has been established, allowing processes to exchange data reliably and sequentially, INDEPENDENT of which systems are communicating or their location in the network."
Values: Unknown

(Output) The ISO.MODEL.COMMUNICATIONS.INTERFACES Unit in MODEL.LIBRARY Knowledge Base

Member slot **PURPOSE** from **MODEL.ELEMENTS**
Inheritance: **UNIQUE.VALUES**
Comment: "description of the purpose the given object is supposed to satisfy or contribute to"
Values: Unknown

Member slot **SESSION.LAYER** from **ISO.MODEL.COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Comment: "fifth lowest of ISO seven-layer network model. Establishes and controls SYSTEM-DEPENDENT aspects of communications sessions between SPECIFIC nodes in the network and bridges the gap between the services provided by the Transport layer and the logical functions running under the operating system in a participating node."
Values: Unknown

Member slot **TRANSMISSION.CHANNEL.TYPE** from **COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
ValueClass: (ONE OF FULL HALF)
Values: Unknown

Member slot **TRANSMISSION.CODE** from **COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
ValueClass: (ONE OF ASCII BCD)
Values: Unknown

Member slot **TRANSMISSION.MODE** from **COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
ValueClass: (ONE OF DIGITAL ANALOG)
Values: Unknown

Member slot **TRANSMISSION.RATE** from **COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
ValueClass: (ONE OF 300 1200 2400 4800 9600 19200 56000)
Comment: "list of line speeds (in bits per second) supported by comm. channel"
Values: Unknown

Member slot **TRANSMISSION.TYPE** from **COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
ValueClass: (ONE OF SYNCHRONOUS ASYNCHRONOUS)
Values: Unknown

Member slot **TRANSPORT.LAYER** from **ISO.MODEL.COMMUNICATIONS.INTERFACES**
Inheritance: **OVERRIDE.VALUES**
Comment: "fourth lowest of ISO seven-layer network model. Provides end-to-end control of communication session once the path has been established, allowing processes to exchange data reliably and sequentially, INDEPENDENT of which systems are communicating or their location in the network."
Values: Unknown

Own slot **DECOMPOSITION.COMPLETE** from **CLASSES**
Inheritance: **UNION**

Figure 4c. MCS AMPE Interfaces Unit (3)

The figures are divided into two halves. The right side shows the ISO Model Communications Interfaces class template from the model library. The left side displays an MCS functional requirement (knowledge base unit) resulting from the customization of the interfaces template. The new unit represents the class of interfaces between MCS sites and (the class of) AMPE exchanges.

Note that data has been filled in for several slot VALUES facets, along with reference documentation and descriptions. In addition, several new slots have been added, which contain specialized information that was not provided for in the generic template. INTERFACE.TYPE, for example, specifies a particular AMPE standard to which contractor interfaces must correspond.

A second slot, MESSAGE.TYPE, was added in because the MCS-to-AMPE communications interface is specifically restricted to message traffic, as opposed to data packets or continuous data streams. This contextual restriction in the application domain necessitates a specification of the types of messages that the AMPE.INTERFACES is expected to support. Note also that the data for the VALUES facet for MESSAGE.TYPE is a structured list, whose form is described in the slot COMMENT facet. Data has been organized into lists because message typing requires three pieces of information for a complete specification - a type name, a format reference, and a description of message handling activities that the interface is expected to support.

Finally, note that many slots (e.g., ISO software protocol layers, FUNCTIONALITY), have not been filled in. Functional requirements models do not generally provide this level of specificity - it is up to the contractor (or developer), in subsequent system model products, to come up with a design that prescribes suitably detailed data. It is also important to realize that data will often be supplied at the superclass or subclass levels (e.g., AUTODIN. INTERFACES, LDMX.INTERFACES). Depending upon the generality and inheritability of the information, it may or may not show up at the AMPE.INTERFACES level. Browsing utilities must reflect this fact, and automatically examine neighboring superclass and subclass units.

3.4.2 Organization of Templates -- Library and System Models

SEIMOAR library templates have been created for a variety of system model elements, including hardware objects (e.g., keyboards), software objects (e.g., compilers), and system or subsystem objects (e.g., communications networks, diagnostic equipment). Templates can be used to encode functionality as well as structural information.

Individual functions and function classes can be characterized in terms of subfunctions and options. Functions can also be organized into hierarchical class taxonomies, reflecting increasing abstraction towards the root class FUNCTIONS.

Figures 5 displays some of SEIMOAR's initial generic functions. Figure 6 displays detailed structure of one subclass of word processing functions. Other broadly useful classes of C^3I functional templates include data base services, system statistics, security services, signal generation, transmission, detection, and processing.

Word processing library templates constitute sources for general purpose user support or utility functions in a requirements model. In the MCS application, for example, word processing functional templates are ingredients in the specification of capabilities for drafting and editing outgoing messages and for responding to incoming messages.

Having identified specific structures and functions for a system, it is important to indicate relationships among these model elements. Figure 7 displays a skeleton tree on which specification units are organized. SEIMOAR's core relations at present are the class-subclass and class-instance relations defined in KEE 2.1. In addition, sequencing and conditional branching relations among functions, typically depicted in functional flow diagrams, are currently encoded in the form of forward-chaining rules. In the MCS test vehicle, for example, these relations prescribe the succession of message processing functions that the AMH applies to incoming or outgoing messages. Sequencing and branching relations constitute important ingredients for the behavioral simulator, as will be seen later.

SEIMOAR associates icons with structural templates, allowing users to construct block diagram illustrations of system architectures. The icons are mousable, with button-activated methods that drive display of internal structures of moused components. For example, given a block diagram display of MCS at a system level (figure 8a), mousing the MCS.WORKSTATION icon activates a display of the internal structures (figure 8b) specified for the workstation (e.g., buses and components). Figure 8c is an iconic display depicting the HOST.PROCESSOR. The mouse function is inherited by the lower level icons for retrieval of further detailed substructures, if they are available in the system model.

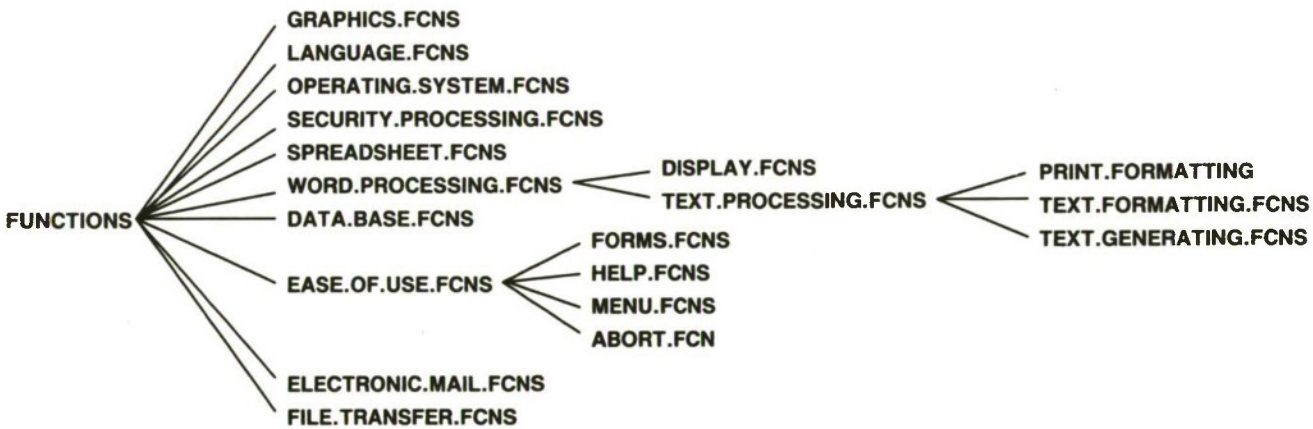

Figure 5. Model Library Generic Functions

Unit: **TEXT.GENERATION.FCNS** in knowledge base **MODEL.LIBRARY**
Created by Adler on 21-Aug-86 16:47:25
Modified by Adler on 22-Aug-86 9:11:09
Superclasses: **TEXT.PROCESSING.FCNS**
Member Of: (**CLASSES** in kb **GENERICUNITS**)

Member slot: **ANIMATE** from **FUNCTIONS**
Inheritance: **METHOD**
ValueClass: (**METHOD** in kb **KEEDATATYPES**)
Values: Unknown

Member slot: **CARRIAGE.RETURN** from **TEXT.GENERATION.FCNS**
Inheritance: **OVERRIDE.VALUES**
Comment: "automatic carriage return and line feed shall be available, as established by a system default model Default line length shall be 69 characters. Options shall include automatic carriage returns that can be set at a designated line lengths"
Values: Unknown

Member slot: **CLOSURE** from **TEXT.GENERATION.FCNS**
Inheritance: **OVERRIDE.VALUES**
Comment: "descriptive text describing object"
Values: "Text processing shall provide capabilities for automatic readjustment of text, characters, words, and lines, so that the resulting text is correctly spaced. The closure function must extend beyond page boundaries and include the automatic creation and deletion of pages as required"
Values: Unknown

Member slot: **CONFIG.ALLOC** from **FUNCTIONS**
Inheritance: **OVERRIDE.VALUES**
Comment: "system hardware and software components that implement this function (or function class)"
Values: Unknown

Member slot: **COPY.TEXT** from **TEXT.GENERATION.FCNS**
Inheritance: **OVERRIDE.VALUES**
Values: Unknown

Member slot: **DESCRIPTION** from **TEXT.GENERATION.FCNS**
Inheritance: **UNIQUE.VALUES**
Comment: "descriptive text describing object"
Values: "A complete set of capabilities for interactive text generation to be used in composing messages, forms, or other data shall be provided. At any point during text generation, users shall be able to scroll or page the text to review previously entered material which is not visible on the screen. Users shall be able to return to the point where the text was being entered prior to such scrolling or paging actions"

Member slot: **INSERT.TEXT** from **TEXT.GENERATION.FCNS**
Inheritance: **OVERRIDE.VALUES**
Comment: "function key shall be provided to enable text insertion to begin at the current cursor location"
Values: Unknown

Member slot: **MERGE.TEXT** from **TEXT.GENERATION.FCNS**
Inheritance: **OVERRIDE.VALUES**
Values: Unknown

Member slot: **MOVE.TEXT.BETWEEN.FILES** from **TEXT.GENERATION.FCNS**
Inheritance: **OVERRIDE.VALUES**
Values: Unknown

Member slot: **MOVE.TEXT.WITHIN.FILES** from **TEXT.GENERATION.FCNS**
Inheritance: **OVERRIDE.VALUES**
Values: Unknown

Member slot: **PURPOSE** from **MODEL.ELEMENTS**
Inheritance: **UNIQUE.VALUES**
Comment: "description of the purpose the given object is supposed to satisfy or contribute to"
Values: Unknown

Member slot: **REPLACE.TEXT** from **TEXT.GENERATION.FCNS**
Inheritance: **OVERRIDE.VALUES**
Values: Unknown

Member slot: **SAVING.TEXT** from **TEXT.GENERATION.FCNS**
Inheritance: **OVERRIDE.VALUES**
Values: Unknown

Member slot: **SEARCH.AND.REPLACE** from **TEXT.GENERATION.FCNS**
Inheritance: **OVERRIDE.VALUES**
Values: Unknown

Member slot: **STRING.SEARCH** from **TEXT.GENERATION.FCNS**
Inheritance: **OVERRIDE.VALUES**
Values: Unknown

Member slot: **TEXT.DELETE** from **TEXT.GENERATION.FCNS**
Inheritance: **OVERRIDE.VALUES**
Comment: "Text processing shall support several screen editing functions. These shall provide for the deletion of a character, word, line, or other areas of text as designated by the cursors position."
Values: Unknown

Member slot: **WORD.WRAPAROUND** from **TEXT.GENERATION.FCNS**
Inheritance: **OVERRIDE.VALUES**
Comment: "For entered text that exceed the set line limit, a whole word wraparound capability shall be provided. Words that cannot fit within the boundaries of the display screen right margin shall automatically wrap around to the start of the left margin on the succeeding line. This capability shall be overridden upon user request (i.e., margin release)"
Values: Unknown

Figure 6. Text Generation Functions Library Template

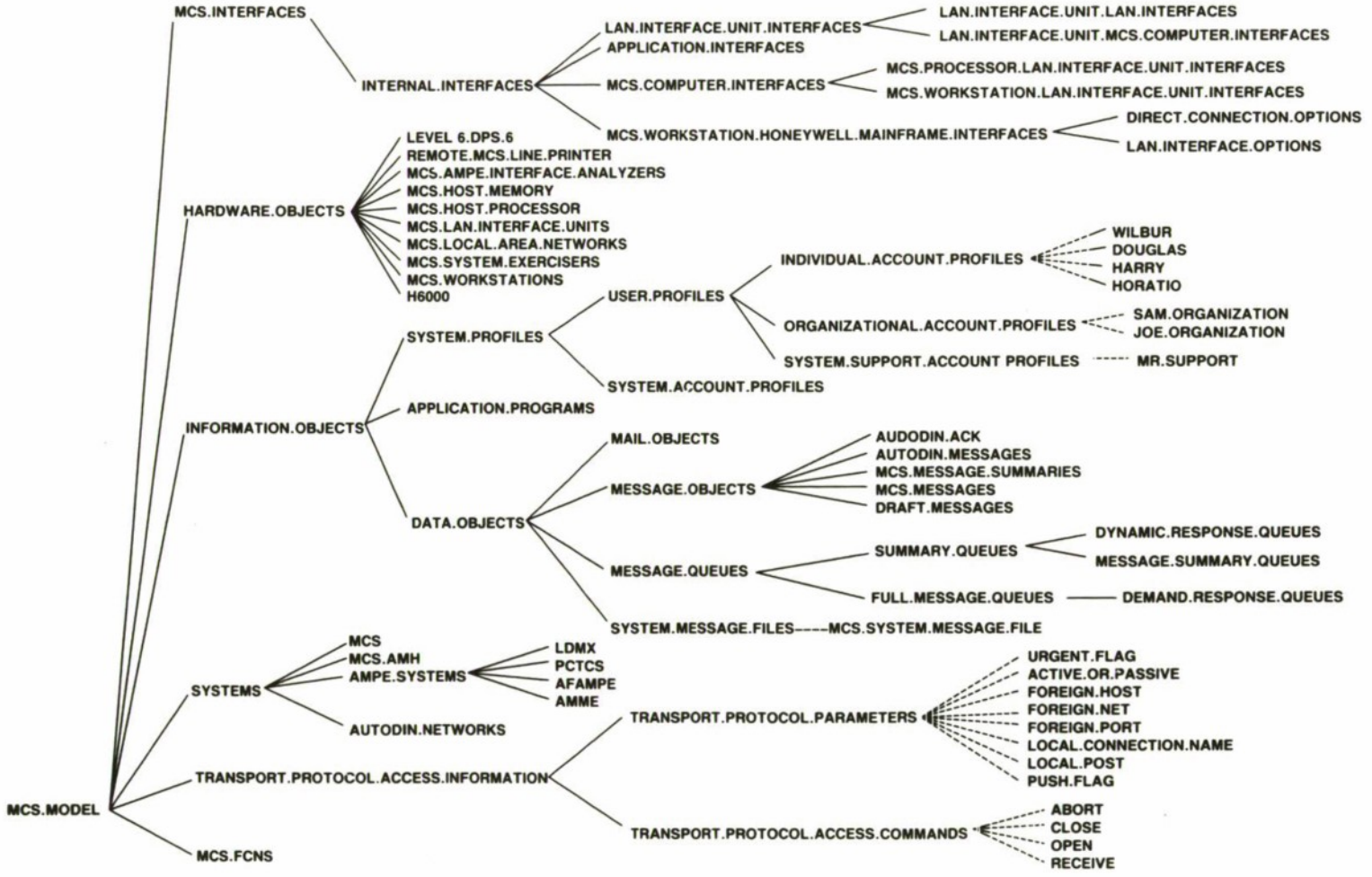

Figure 7. MCS System Model Tree Graph

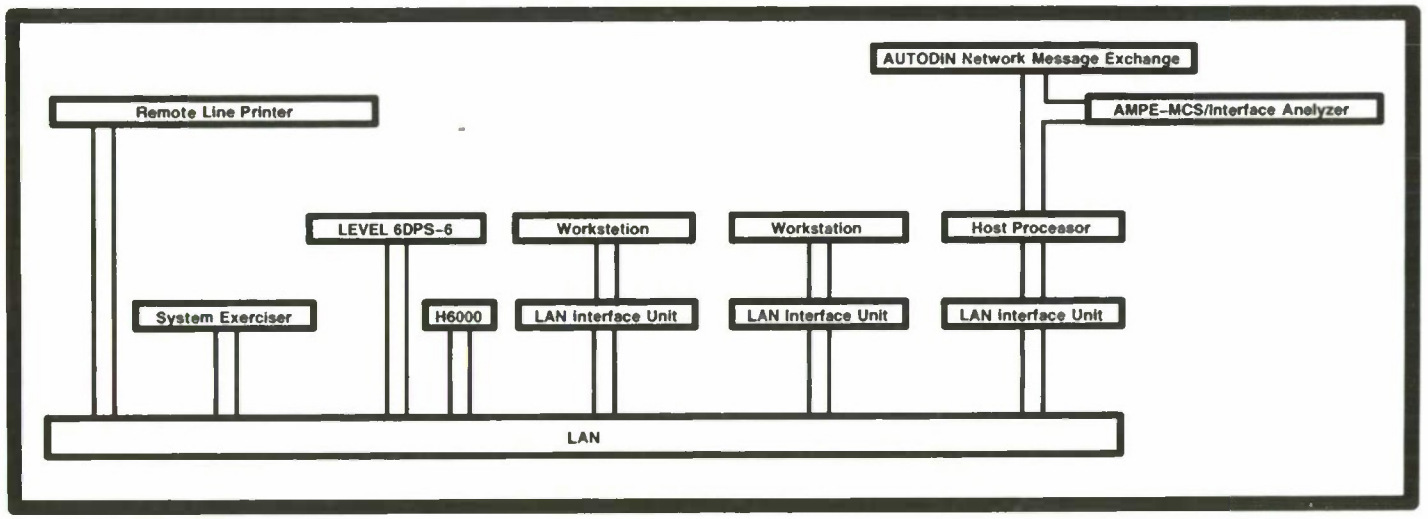

Figure 8a. MCS System Level Block Diagram

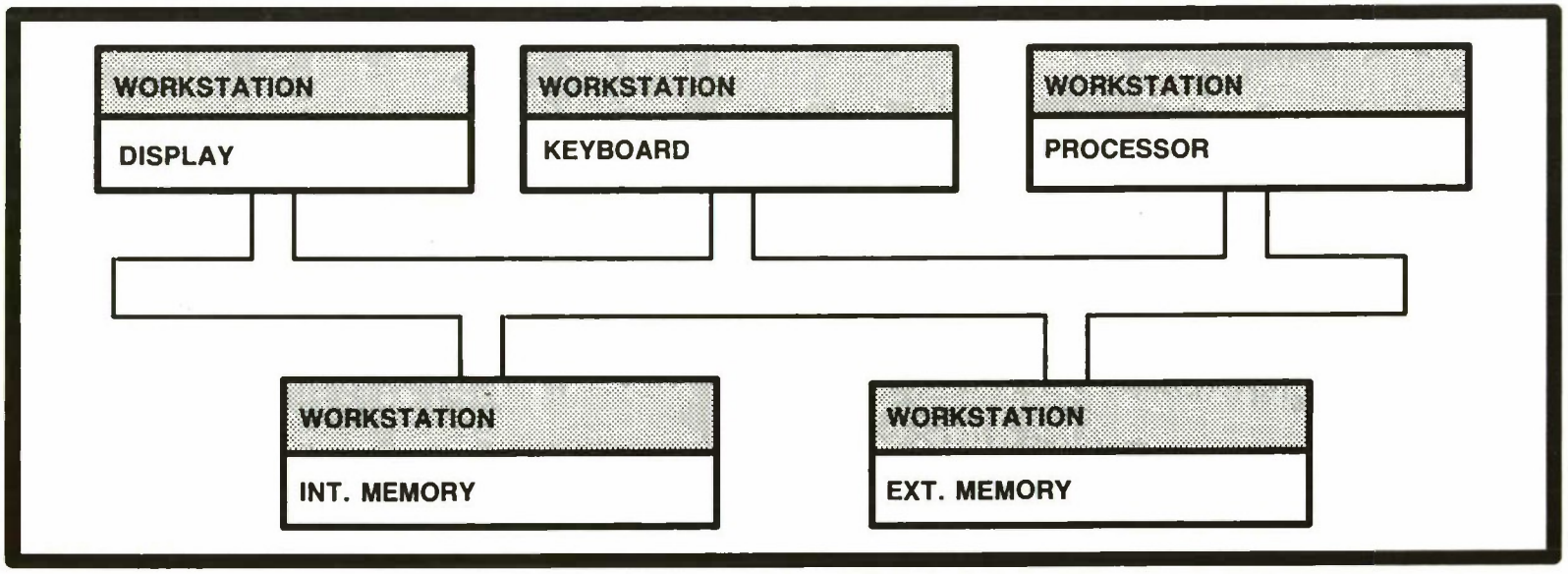

Figure 8b. MCS Workstation Block Diagram

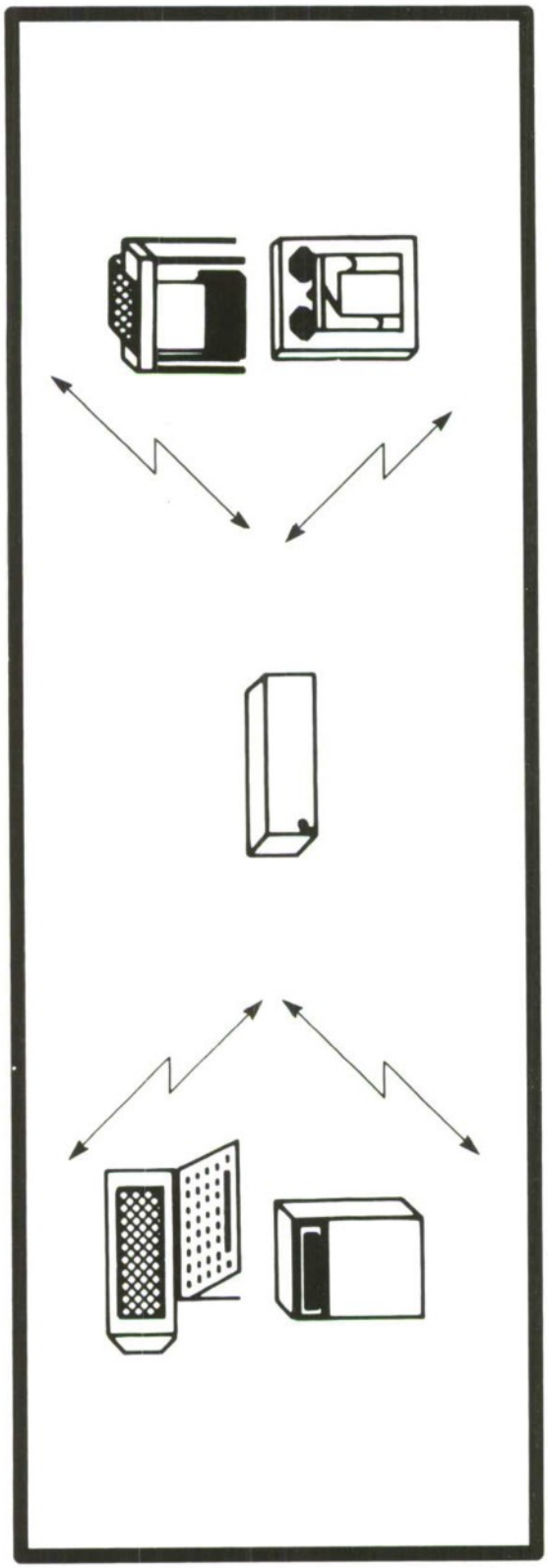

Figure 8c. Host Processor Iconic Diagram

3.5 BEHAVIORAL SIMULATION -- ARCHITECTURE AND EXAMPLES

In acquisition contexts, behavior refers to the collections of event sequences that take place within systems, which realize specific functional capabilities. That is, functions act to induce one or more changes of state into the system, its component structures, and the data objects that the system is supposed to be processing.

SEIMOAR incorporates a discrete time functional simulator to support dynamic modeling of system behaviors. Currently, SEIMOAR's simulator is a custom-built object-oriented shell that features a synchronous clock mechanism suitable for modeling sequential execution of functions. The simulator is implemented by a set of frame units and attached utility methods in the modeling library.

Figure 9 illustrates the architecture for SEIMOAR's simulator shell. The simulator is run by (object-oriented style) methods, activated by messages sent out from shell units representing a system clock and a centralized activity handler.

When a new application model knowledge base is created, this shell is incorporated into that system description. The user supplies two forward-chaining rule bases and a collection of function frames incorporating methods for simulating system behavior to reflect functional actions. The rule bases and the functions are the application-specific items that are needed to characterize system behaviors to the simulation shell. Figure 10 displays some of the frames representing MCS AMH functions.

Briefly, an off-line scenario generator allows users to build test exercises. Each test scenario consists of a set of events to be injected into the system model, at predetermined times. The injection of events causes the system model to respond with a set of programmed behaviors, corresponding to the ordered actions of system functions.

For the MCS application, the generator constructs a test scenario, consisting of a collection of (model) incoming messages from the AUTODIN network. Messages can be partitioned into multiple segments, which can be interspersed, noncontiguously, with other messages. AUTODIN messages have a precedence attribute (i.e., relative urgency), and a variety of other message fields (e.g., keywords, text, originator, subject), as shown in figure 11.

Aside from a simple identifying label, the critical event attribute required to drive the simulator is the TIME.OF.ARRIVAL slot, which indicates the injection time into the simulation.

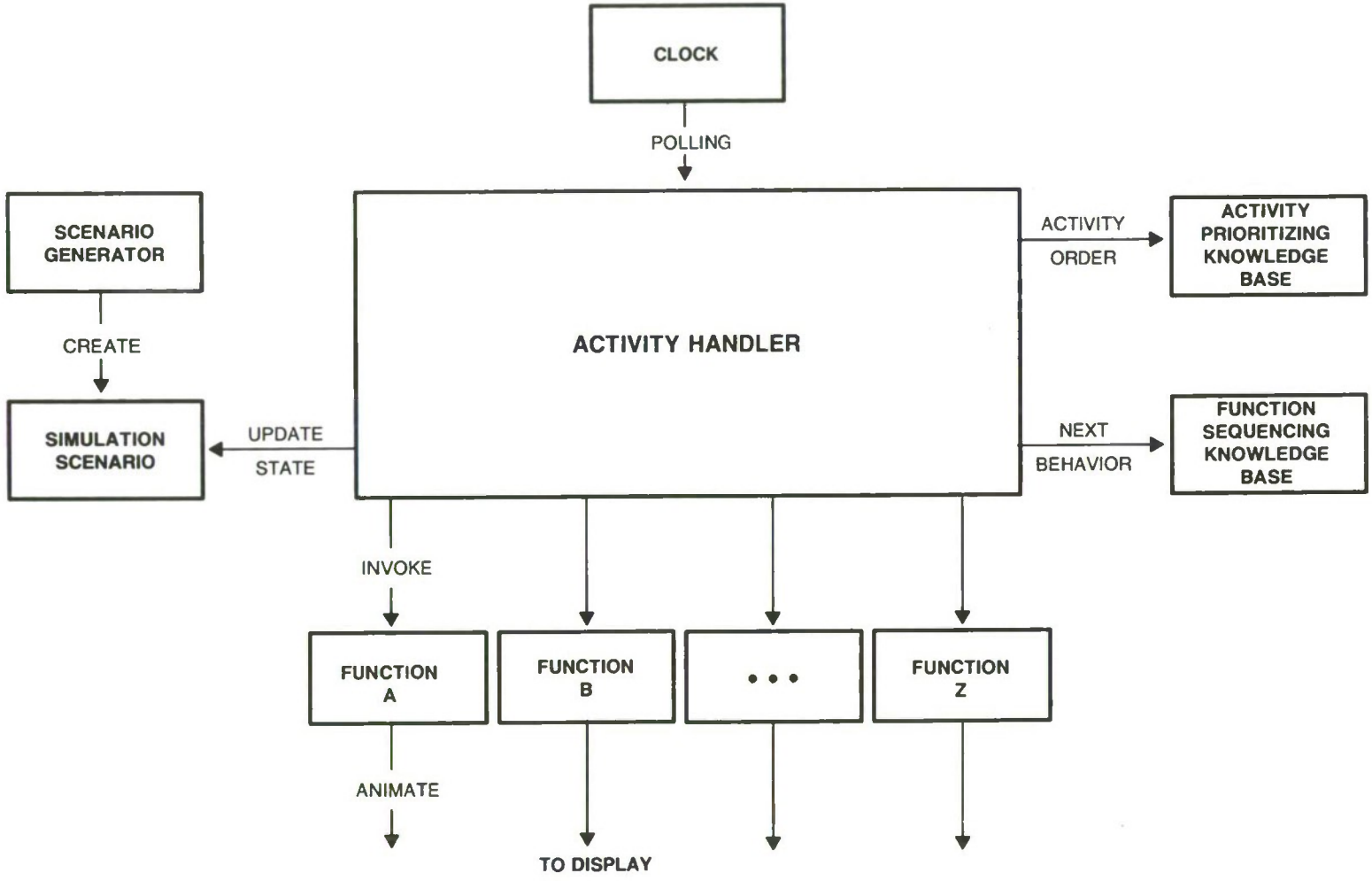

Figure 9. SEIMOAR Behavioral Simulation Shell

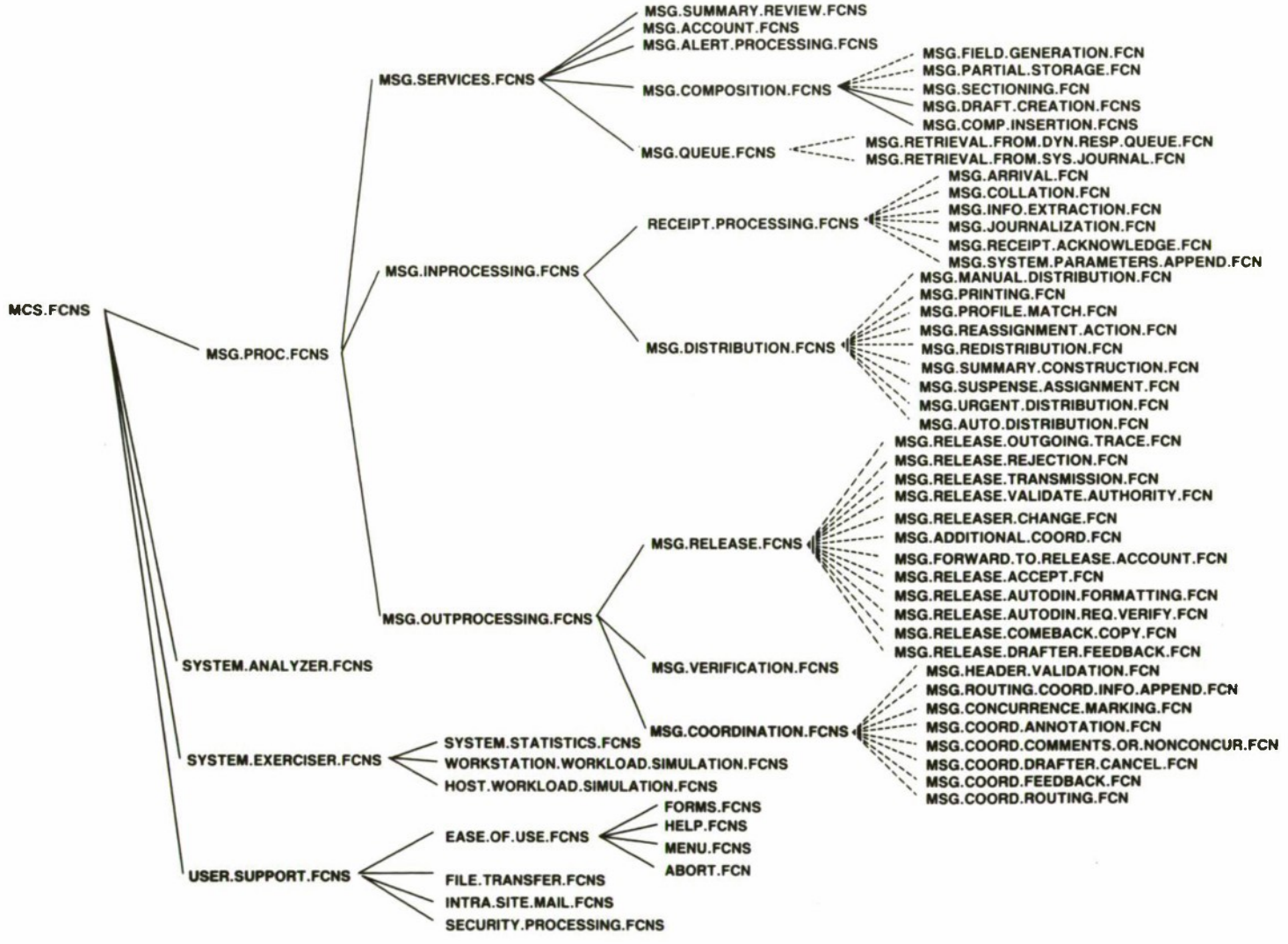

Figure 10. MCS AMH Function Units

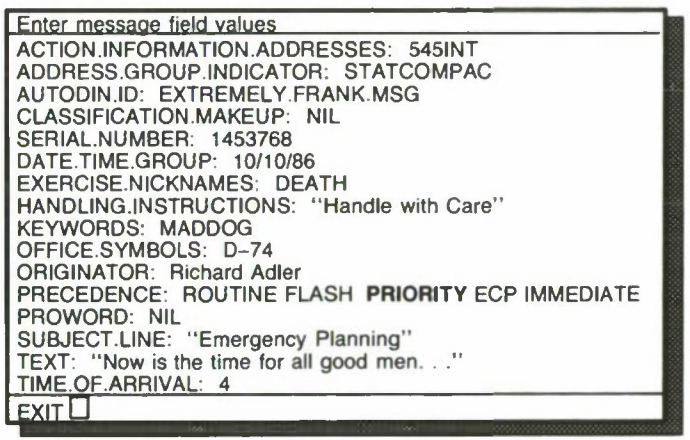

Figure 11. AUTODIN Message Unit Fields

Because the simulator models sequential processes, the generator monitors TIME.OF.ARRIVAL slot data for test events and ensures unique injection times for each event belonging to a given test scenario. The test vehicle MCS AMH functional requirements model also requires that events (i.e., arriving message objects), have an AUTODIN precedence slot value assignment. The generator enforces this constraint by blocking event creation until a value is supplied, and by issuing a suitable explanatory warning to the user.

The main control loop method, activated through a SEIMOAR main menu selection, cycles a global clock (unit) that polls the activity handler. The activity handler has its own poll cycle method, which basically consists of four message calls. The first message is sent to the simulation scenario. If the scenario contains any events whose time of arrival matches the global clock time, those events are removed from the scenario and injected into the activity handler system state slot. This slot is simply a list consisting of all the objects (e.g., signals, messages), currently being processed by the system model.

The second and third message calls invoke two forward-chaining rule bases. The first rule base encodes an algorithm describing the prioritization behavior of the activity handler. In other words, this rule base figures out which object in the activity handler system state slot has the highest processing priority.

In the MCS application, prioritization consists of three rules (figure 12), which in turn call upon custom-written LISP list processing functions. The rules are weighted and written using dummy variables in such a fashion as to guarantee a specific sequence of rule firings. The net result is to identify the message unit currently in the AMH with the highest urgency for processing, based on time of arrival, designated message precedence, and current processing state of the message.

The third method takes the highest priority object and determines the appropriate processing to perform on it by calling another forward-chained rule base. This second knowledge base contains rules that encode functional flows, the pattern of sequencing and conditional branching that determines what action to perform next on the given object. This knowledge base amounts to a functional state transition network.

Several sample rules for the MCS AMH system are shown in figure 13. The initial antecedent clause is used simply to bind a dummy variable to the appropriate message object unit. Determination of the next function to apply to an object is based on the previous function and on the message's state. Some of these state variables

Member slot: **ASSUMPTIONS** from **MSG.PRIORITIZING.RULES**
Inheritance: **OVERRIDE.VALUES**
Comment: "list of presuppositions underlying behavioral model - processing is assumed to be sequential, wherein one function is performed on one object at any one time. The scenario generator is constructed so as to ensure that one and only one event (the arrival of an autodin.msg or the creation of a draft.msg) takes place at any given instant (time.of.arrival). The prioritizing rules also presuppose that the priority of autodin.msgs is known to the object.handler upon arrival, before any field.extraction functions have been applied. This error, present in the A-spec source implicitly, is left in the model for demonstration purposes. This rule base also makes explicit the implicit prioritizing principle that given multiple objects of equal urgency, the object with the earliest time is to be given precedence."
Values: Unknown

```
(COMMENT "root rule class for message.handler message prioritizing rule base")
(RULES.IN.CLASS
  (COLLECT.HIGHEST.PRIORITY.CANDIDATES.RULE
    (IF
      (AND (GET.VALUES 'MH.HANDLER.OBJECT 'ALL.CANDIDATE.OBJECTS)
           (NULL (GET.VALUES 'MH.HANDLER.OBJECT 'HIGHEST.PRECEDENCE.CANDIDATE.OBJECTS)))
      THEN
      (PUT.VALUES
       'MH.HANDLER.OBJECT
       'HIGHEST.PRECEDENCE.CANDIDATE.OBJECTS
       (COLLECT.HIGHEST.PRIORITY.CANDIDATES (GET.VALUES 'MH.HANDLER.OBJECT
                                                        'ALL.CANDIDATE.OBJECTS)))))
  (COLLECT.OBJECTS.NEEDING.RCPT.PROCESSING.RULE
    (IF
      (NULL (GET.VALUES 'MH.HANDLER.OBJECT 'ALL.CANDIDATE.OBJECTS))
      THEN
      (PUT.VALUES
       'MH.HANDLER.OBJECT
       'ALL.CANDIDATE.OBJECTS
       (COLLECT.OBJECTS.NEEDING.PREFERRED.PROCESSING (GET.VALUES 'MH.HANDLER.OBJECT
                                                                 'HANDLER.STATE)))))
  (FIND.EARLIEST.TOA.CANDIDATE.RULE
    (IF
      (AND (GET.VALUES 'MH.HANDLER.OBJECT 'HIGHEST.PRECEDENCE.CANDIDATE.OBJECTS)
           (NULL (GET.VALUES 'MH.HANDLER.OBJECT 'NEXT.OBJECT.TO.PROCESS)))
      THEN
      (PUT.VALUES
       'MH.HANDLER.OBJECT
       'NEXT.OBJECT.TO.PROCESS
       (FIND.EARLIEST.TOA.CANDIDATE (GET.VALUES 'MH.HANDLER.OBJECT
                                                'HIGHEST.PRECEDENCE.CANDIDATE.OBJECTS))))))
```

Figure 12. AMH Message Prioritization Rules

```
(MSG.INPROCESSING.TERMINATION.RULE

  (IF (AND (EQUAL ?MSG
                  (FIRST (GET.VALUES 'MH.HANDLER.OBJECT 'NEXT.OBJECT.TO.PROCESS)))
           (OR (THE CURRENT.PROCESSING.STATE OF ?MSG IS MSG.MANUAL.DISTRIBUTION.FCN)
               (THE CURRENT.PROCESSING.STATE OF ?MSG IS MSG ACCOUNT.FCNS)))
      THEN
      (THE NEXT.PROCESSING.STATE OF ?MSG IS 'COMPLETION)))

(MSG.JOURNALIZATION.RULE

  (IF (AND (EQUAL ?MSG
                                  (FIRST (GET.VALUES 'MH.HANDLER.OBJECT
                                                     'NEXT.OBJECT.TO.PROCESS)))
                             (THE CURRENT.PROCESSING.STATE
                                  OF
                                  ?MSG
                                  IS
                                  MSG.SYSTEM.PARAMETERS.APPEND.FCN))
                        THEN
                        (THE NEXT.PROCESSING.STATE
                             OF
                             ?MSG
                             IS
                             MSG.JOURNALIZATION.FCN)))

(MSG.MANUAL.DISTRIBUTION.RULE

  (IF (AND (EQUAL ?MSG
                  (FIRST (GET.VALUES 'MH.HANDLER.OBJECT 'NEXT.OBJECT.TO.PROCESS)))
             (THE CURRENT.PROCESSING.STATE OF ?MSG IS MSG.SUMMARY.CONSTRUCTION.FCN)
             (THE AUTOMATICALLY.DISTRIBUTABLEP OF ?MSG IS F))
      THEN
      (THE NEXT.PROCESSING.STATE OF ?MSG IS MSG.MANUAL.DISTRIBUTION.FCN)))

(MSG.PROFILE.MATCH.RULE

  (IF (AND (EQUAL ?MSG
                                  (FIRST (GET.VALUES 'MH.HANDLER.OBJECT
                                                     'NEXT.OBJECT.TO.PROCESS)))
                             (THE CURRENT.PROCESSING.STATE
                                  OF
                                  ?MSG
                                  IS
                                  MSG.RECEIPT.ACKNOWLEDGE.FCN))
                        THEN
                        (THE NEXT.PROCESSING.STATE
                             OF
                             ?MSG
                             IS
                             MSG.PROFILE.MATCH.FCN)))
```

Figure 13. AMH Function Sequencing Rules (Examples)

(e.g., whether to activate manual, automatic, or urgent message distribution modes of processing), depend on changes of state effected by preceding functions (e.g., profile matching function).

The fourth message call causes actual functional behavior in the system model to be performed. Specifically, the message activates a method associated with the particular function selected by the function sequencing knowledge base to be applied to the object selected by the object prioritization knowledge base.

Methods are supplied by SEIMOAR users assembling functional requirements or other system description models. Each method, consisting of LISP and KEE functions, induces state changes in the simulator, the system model, and data object units: new objects can be created; objects can be inserted or removed from the activity handler system state slot; and simulator, system component, and data object attributes can be altered. All of these changes correspond to the sequence of events that represent the implementation of the actions of particular functions.

Code characterizing the behavior for a typical system function, acknowledging AMH receipt of a message back to the AUTODIN, is displayed in figure 14a. Figure 14b displays the code that drives the iconic animation of the behavioral simulation. The action of the receipt acknowledgment function in the MCS AMH functional requirements system model is depicted through the creation of a special acknowledge object, which represents the transmission, along the MCS-AMPE interface back to the AUTODIN, of successful message receipt. The new object is named by appending a string to the received message name. A message to the new object tracer monitor is sent, along with KEE function calls that update the states of simulator and data objects. Figure 14c depicts a snapshot of the iconic animation sequence for this function.

Basically, the simulator reflects changes of state in model data objects or system elements in terms of changes to KEE unit slot VALUES data items. Complicated behaviors are readily captured, as can be illustrated by sketching the sequence of modeling actions for AMH message distribution. Model AUTODIN message units have slots for message ID, keywords, exercise nicknames, and other descriptors. The MCS AMH model contains several model user and system support accounts, each containing its own descriptors. The profile match function compares AUTODIN message keyword and exercise nickname field contents against account values. If matches occur, flags are set that drive subsequent message distribution activities, manual, automatic, or urgent routing. Model user account message summary queues are created and filled, as appropriate. Figures 15a and b display simulation methods for message profile matching and

```
Own slot: INVOKE from MSG.RECEIPT.ACKNOWLEDGE.FCN
  Inheritance: METHOD
  ValueClass: (METHOD in kb KEEDATATYPES)
  Comment: "method used by behavioral simulator to alter the design model in correspondence with the changes of system state induced by performing the given function"
  Values: |MCS>MSG.RECEIPT.ACKNOWLEDGE.FCN::INVOKE!method|
(LAMBDA (THISUNIT $OBJECT)
  (LET (($NU.ID (READ-FROM-STRING (STRING-APPEND $OBJECT ".autodin.ack"))))
    (CREATE.UNIT $NU.ID NIL NIL AUTODIN.ACK)
    (UNITMSG '(UTILITY.METHODS MCS) 'UPDATE.NEW.OBJECTS $NU.ID)
    (PUT.VALUE $NU.ID MSG.ID $OBJECT)
    (PUT.VALUE $NU.ID 'CURRENT.PROCESSING.STATE (GET.VALUE $OBJECT 'NEXT.PROCESSING.STATE))
    (PUT.VALUE $OBJECT 'CURRENT.PROCESSING.STATE (GET.VALUE $OBJECT 'NEXT.PROCESSING.STATE))
    (PUT.VALUE 'MH.HANDLER.OBJECT 'CURRENT.PROCESSING.STATE (GET.VALUE $OBJECT 'NEXT.PROCESSING.STATE))))
```

Figure 14a. Function Simulation Method


```
Own slot: ANIMATE from MSG.RECEIPT.ACKNOWLEDGE.FCN
  Inheritance: METHOD
  ValueClass: (METHOD in kb KEEDATATYPES)
  Values: |MCS>MSG.RECEIPT.ACKNOWLEDGE.FCN::ANIMATE!method|
(LAMBDA (THISUNIT)
  (LET* ((PANEL (GET.VALUE 'SHELL MCS) 'ILLUSTRATION.PANEL))
    (PANEL-WINDOW (GET.VALUE PANEL 'WINDOW))
    (PANEL-REGION (GET.ACTIVE.IMAGE.REGION PANEL))
    (X (+ (FIRST PANEL-REGION) 454))
    (Y (+ (SECOND PANEL-REGION) 120))
    (LENGTH 32)
    (NEWMSG NIL))
  (BITBLT* IMAGE.PANEL.MCS.COMPONENTS.BM 0 0 PANEL-WINDOW 0 0 607 190)
  (SETQ NEWMSG (UNITMSG '(ICON.AUTODIN.ACK MCS.IMAGES) 'CREATE.PROGRAMMATICALLY (CREATE.UNIT (GENSYM 'A) 'MCS NIL
    '((AUTODIN.ACK MCS))) (LIST X Y 8 8) NIL 'EXPAND PANEL))
  (UNITMSG '(ANIMATION.UTILITY METHODS MCS.IMAGES) 'MOVE.MSG.VERT.UP NEWMSG X Y LENGTH)
  (PUT.VALUE '(SHELL MCS) 'CURRENT.BITMAP 'IMAGE.PANEL.MCS.COMPONENTS.BM)))
```

Figure 14b. Function Animation Method

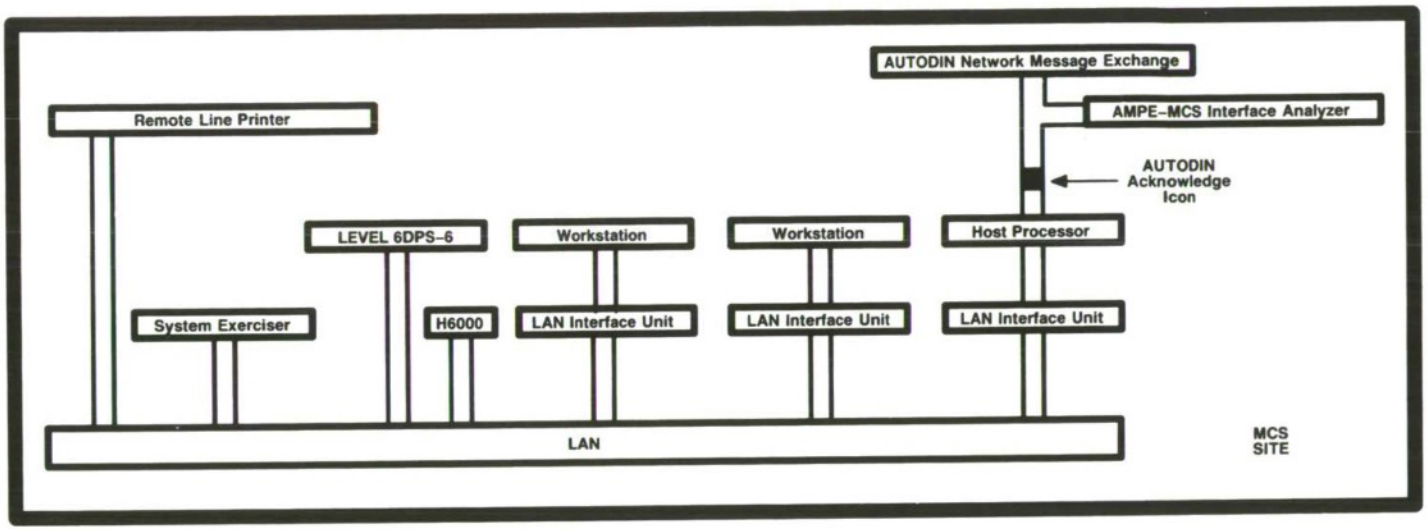

Figure 14c. Animation Snapshot

The INVOKE Slot of the MSG.PROFILE.MATCH.FCN Unit

```
Own slot: INVOKE from MSG.PROFILE.MATCH.FCN
  Inheritance: METHOD
  ValueClass: (METHOD in kb KEEDATATYPES)
  Prog.Notes: "only exercise.nicknames and keywords are currently matched in invoke"
    Facet Inheritance: OVERRIDE.VALUES
  Comment: "method used by behavioral simulator to alter the design model in correspondence with the changes of system state induced by performing the given function"
    Facet Inheritance: OVERRIDE.VALUES
  Values: |MCS>MSG.PROFILE.MATCH.FCN::INVOKE!method|
(LAMBDA (THISUNIT $OBJECT)
    (MAPCAR (LAMBDA ($X)
              (COND ((OR (INTERSECTION (GET.VALUES $OBJECT 'KEYWORDS) (GET.VALUES $X'KEYWORDS))
                         (INTERSECTION (GET.VALUES $OBJECT 'EXERCISE.NICKNAMES) (GET.VALUES $X EXCERCISE.NICKNAMES)))
                (ADD.VALUE $OBJECT 'DISTRIBUTION.LIST $X))
                (T NIL)))
        (MAPCAR 'UNIT.NAME (UNIT.ALLCHILDREN 'USER.PROFILES 'MEMBER)))
    (COND ((MEMQ (GET.VALUE $OBJECT 'PRECEDENCE) '(ECP FLASH))
      (PUT.VALUE $OBJECT 'URGENTP 'T))
      (T (PUT.VALUE $OBJECT 'URGENTP 'F)))
    (COND((NULL (GET.VALUES $OBJECT 'DISTRIBUTION.LIST))
      (PUT.VALUE $OBJECT 'AUTOMATICALLY.DISTRIBUTABLEP 'F))
    (T (PUT.VALUE $OBJECT 'AUTOMATICALLY.DISTRIBTABLEP 'T)))
  (PUT.VALUE $OBJECT 'CURRENT.PROCESSING.STATE (GET.VALUE $OBJECT 'NEXT.PROCESSING.STATE))
  (PUT.VALUE 'MH.HANDLER.OBJECT 'CURRENT.PROCESSING.STATE (GET.VALUE $OBJECT 'NEXT.PROCESSING.STATE)))
```

Figure 15a. Behavioral Simulation Message Profile Match Function

(Output) The INVOKE Slot of the MSG.AUTO.DISTRIBUTION.FCN Unit

```
Own slot: INVOKE from MSG.AUTO.DISTRIBUTION.FCN
  Inheritance: METHOD
  ValueClass: (METHOD in kb KEEDATATYPES)
  Comment: "method used by behavioral simulator to alter the design model in correspondence with the changes of
system state induced by performing the given function"
    Facet Inheritance: OVERRIDE.VALUES
  Values: |MCS>MSG.AUTO.DISTRIBUTION.FCN::INVOKE!method|
(LAMBDA (THISUNIT $OBJECT)
  (LET (($DIST.LIST (GET.VALUES $OBJECT 'DISTRIBUTION.LIST)))
    (FORMAT T "Message ~ A will be distributed automatically to users ~A" $OBJECT $DIST.LIST))
  (LOOP FOR
    $NAME
    IN
    $DIST.LIST
    DO
    (COND ((UNIT.EXISTS.P (READ-FROM-STRING (STRING-APPEND "MSG.SUMMARY.QUEUE." $NAME)))
      ADD.VALUE (READ-FROM-STRING (STRING-APPEND "MSG.SUMMARY.QUEUE." $NAME))
      NORMAL.DISTRIBUTION.MSGS $OBJECT))
      (T
      (CREATE.UNIT (READ-FROM-STRING (STRING-APPEND "MSG.SUMMARY.QUEUE." '$NAME)) MCS NIL
      (MESSAGE.SUMMARY.QUEUES))
      (PUT.VALUE (READ-FROM-STRING (STRING-APPEND "MSG.SUMMARY.QUEUE." $NAME)) 'NORMAL.DIS-
      TRIBUTION.MSGS $OBJECT))))
(PUT.VALUE $OBJECT 'CURRENT.PROCESSING.STATE (GET.VALUE $OBJECT 'NEXT.PROCESSING.STATE))
(PUT.VALUE 'MH.HANDLER.OBJECT 'CURRENT.PROCESSING.STATE (GET.VALUE $OBJECT 'NEXT.PROCES-
SING.STATE)))
```

Figure 15b. Behavioral Simulation Message Automatic Distribution Function

automatic distribution functions. Similarly detailed behaviors are established for collation of multiple AUTODIN message segments and other complex functions.

The generality of a simulator shell oriented toward modeling individual functions makes it possible to develop behavioral models at all prescribed levels of design description, modeling system, component, and subcomponent processes as appropriate. In normal functional requirements, for example, behavior is specified fairly abstractly, in terms of changes of state at a system level (e.g., moving from completion of one function to the next prescribed action). Functional specifications are deliberately written to minimize presuppositions concerning particular design implementations in order to leave contractors as free as possible to devise their own architectural strategies. Subsequent specifications and design descriptions characterize functional actions at a much finer-grained level, in terms of changes of state of data objects and individual hardware and software components.

Figure 16 shows the simulator trace, a collection of windows that provide a visual interface to simulator events. The current test scenario name is shown, together with the clock cycle, current data object (message) being handled, current function, new objects created during the simulation run, and a general trace of activity. An interrupt monitor allows simulator action to be suspended at any clock pulse, whereupon the unfinished scenario events can be edited or the knowledge base containing the system model and active data objects can be browsed.

The SEIMOAR simulator provides the capabilities to reflect modeled behaviors visually in a window displaying icon animation sequences. Graphic utility methods stored in the model library move icons signifying data objects across the screen horizontally or vertically between icons representing system model structures.

For example, in the test vehicle system model, message arrival is represented by a box icon (incoming message) displayed initially below the AMPE.EXCHANGE icon and then reappearing successively lower along a channel icon (MCS.AMPE.INTERFACE) before disappearing at the HOST.PROCESSOR icon. At this point, a bitmap showing the internal structure of that component is displayed, and the icon traverses a bus before disappearing at the HOST.PROCESSOR.SYSTEM.MEMORY icon. The capability to cycle through bitmaps and animation sequences, which corresponds conceptually to changing levels of descriptive detail, is extremely useful in illustrating complex behaviors.

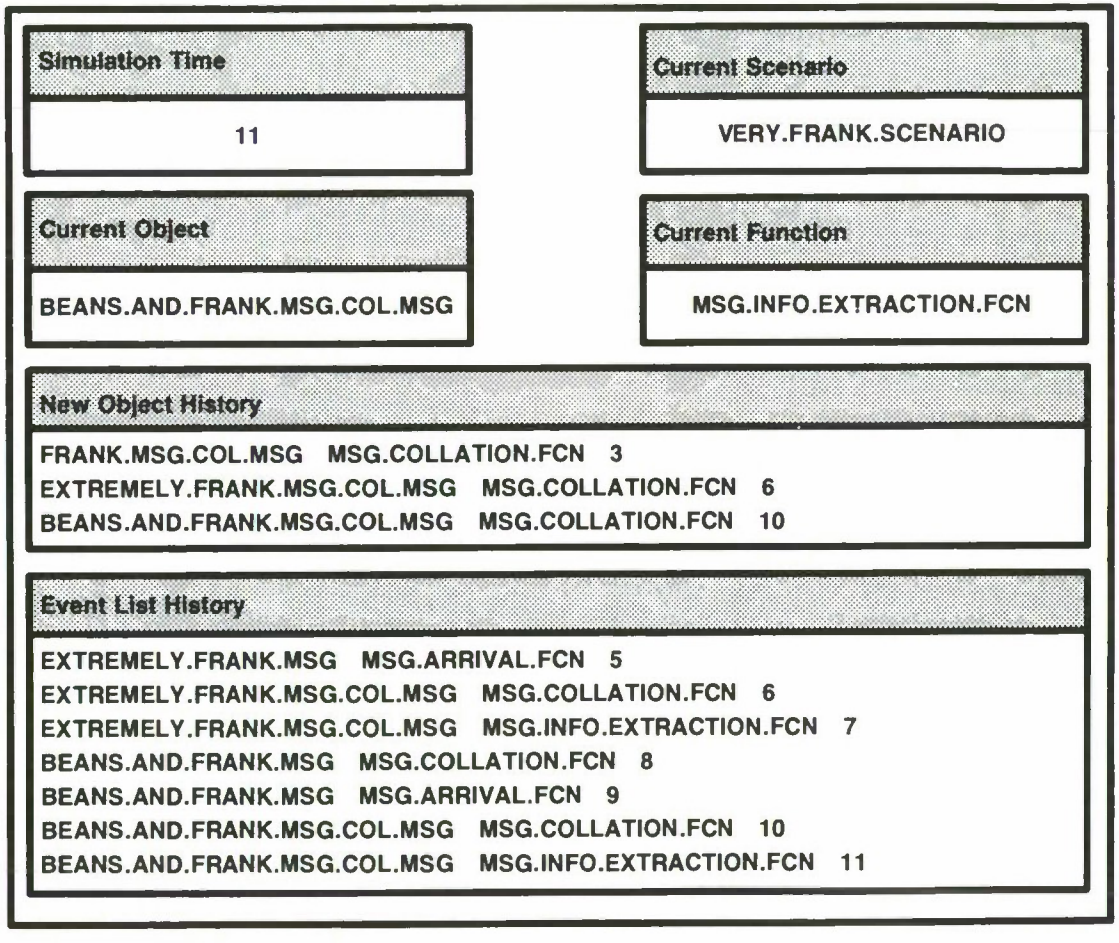

Figure 16. AMH Behavioral Simulation Trace

3.6 SIMULATOR EXTENSIBILITY

An attractive feature of the object-oriented simulator architecture is its extensibility. The simulator shell can easily be modified to support nested activity models. The clock would poll a super-handler unit on each time cycle, which then decides which activity handler to activate, which then determines and executes a function.

Rather than modeling the AMH alone, for example, the super-handler could model the HOST.PROCESSOR, which might activate either the AMH handler, a message statistics handler, or some other application resident in the host.

This extension could be implemented through a simple modification to the clock polling method: the addition of a super handler unit and other handlers, and the addition of a knowledge base coding the super-handler prioritization control. All handlers determine their own activities, through reasoning based on purely internal structures (viz., prioritization and sequencing knowledge bases). The superhandler prioritization knowledge base, like the (AMH) handler control knowledge base, reflects the operational behavior of that system element.

3.7 DISCOVERING REQUIREMENTS ERRORS USING THE SIMULATOR

Although there is no intelligent support for dynamic analysis at present, the process of describing and simulating functional behavior provides an excellent consistency check on model characterizations. Two examples from modeling experiments with the MCS AMH functional requirements are relevant here.

The message prioritization algorithm specified in the requirement assigns preferential weighting for messages requiring receipt processing, higher AUTODIN precedences, and FIFO queueing. On simulating functional behaviors, it was found that the system would not know AUTODIN precedences of messages until the information extraction function was executed. However, the prioritization algorithm assumed that the AMH knew this information upon message arrival, rather than three functions later. In other words, the prioritization algorithm was inconsistent with actual function sequencing.

A second problem surfaced in coding function sequencing from the requirements functional flowchart. It turned out that the flowchart combined two incompatible points of view: all automated message processing functions are depicted from the point of view of

a message proceeding through a prescribed flow sequence. Once messages were distributed to users, however, the flow chart shifted to the point of view of a user selecting functions to perform on individual messages stored in their summary queue. This second perspective, thus, is of a user acting on multiple messages. An explicit user model is required to simulate the AMH capabilities here, and the requirements omitted any description of users, their needs, and likely behaviors. Moreover, implicit heterogeneity such as this constitutes poor design methodology for flowcharts.

SECTION 4

SEIMOAR FOLLOW-ON WORK

Work to date on SEIMOAR has been of a proof-of-concept nature, exploring the functional capabilities of knowledge-based acquisition support environments. Functional and structural information for an unusually detailed C^3I system functional specification was captured successfully using the template approach. The behavior of the same test vehicle was successfully modeled using a generalized discrete time functional simulator. Ideas for constructing a highly visual user interface to structural and behavioral information using icons and iconic animation were validated. Most importantly, all of these capabilities were derived from a single, uniform representational framework, and integrated within a single development environment, in line with the initial model-based strategy described earlier.

However, the resulting system is rather fragmentary, skeletal, and of limited utility to its intended audience. This is typical of proof-of-concept system experiments. The objective for subsequent effort is to elaborate the current SEIMOAR system into a full-scale acquisition support prototype. The specific goals of next year's work are twofold: first, to extend the functional capabilities of the tool, and second, to develop a coherent interface that is suitable for a general acquisition support user community. It is hoped that within 24 months, the prototype will be taken as the basis for a functional specification for a production system to be let out on contract.

A significant amount of effort has already gone into the planning and design of both the functional enhancements and user interface. Time and staff constraints prevented actual implementation in the initial project phases. Nevertheless, a review of current plans and design strategies will help to convey a picture of the potential of the SEIMOAR system in acquisition support.

The review is divided into three parts. The first section covers extensions to the basic representational framework. In addition, new static analysis functions are discussed. These extend the acquisition support capability well beyond simple generation of system models. The second section describes planned extensions to SEIMOAR's simulator and dynamical analysis functional enhancements. The third and final section covers development plans for a

comprehensive user interface to SEIMOAR. The interface is intended to make the tool and its capabilities accessible to a general acquisition support community.

4.1 STRUCTURAL AND FUNCTIONAL REPRESENTATION EXTENSIONS

4.1.1 Filling in the Model Library

The MCS functional requirements modeling exercise helped to validate the library-based strategy for system description construction in principle. The effectiveness of copy-and-edit strategies in practice depends directly on the richness of the available store of generic C^3I templates. Little if any gain in productivity is achieved if system model creators are forced to generate more than a few abstracted structural and functional templates.

It is very important, therefore, to populate the library with a good initial stock of C^3I structural and functional templates. The overall structure of SEIMOAR's library is depicted in figure 17. At present, the library itself resembles more of a card catalog than the library: most of the unit classes contain few if any attributes and relations. Current plans are to conduct intensive interviews with acquisition support specialists, in order to define and characterize suitable library elements.

Some of the templates shown in the figure are reserved for future experimentation on automated reasoning. For example, a branch of the library taxonomy contains units storing information that will regulate reasoning across different levels of description. The idea is that template unit slots can be tagged with facets which type the associated attribute or relation as belonging to one or more levels of description. Reasoning within or across specific levels can then automatically be restricted to appropriate slots, by mechanically sorting slots by facet values.

4.1.2 Configuration Allocation, Function Traceability and Structural Decomposition

Representational capabilities will have to be extended if SEIMOAR is to support acquisition activities through preliminary design descriptions (STLDD). Two features of particular importance are traceability and configuration allocation. Automation of traceability information is one of the most urgent needs in acquisition efforts for large systems. Traces must first be

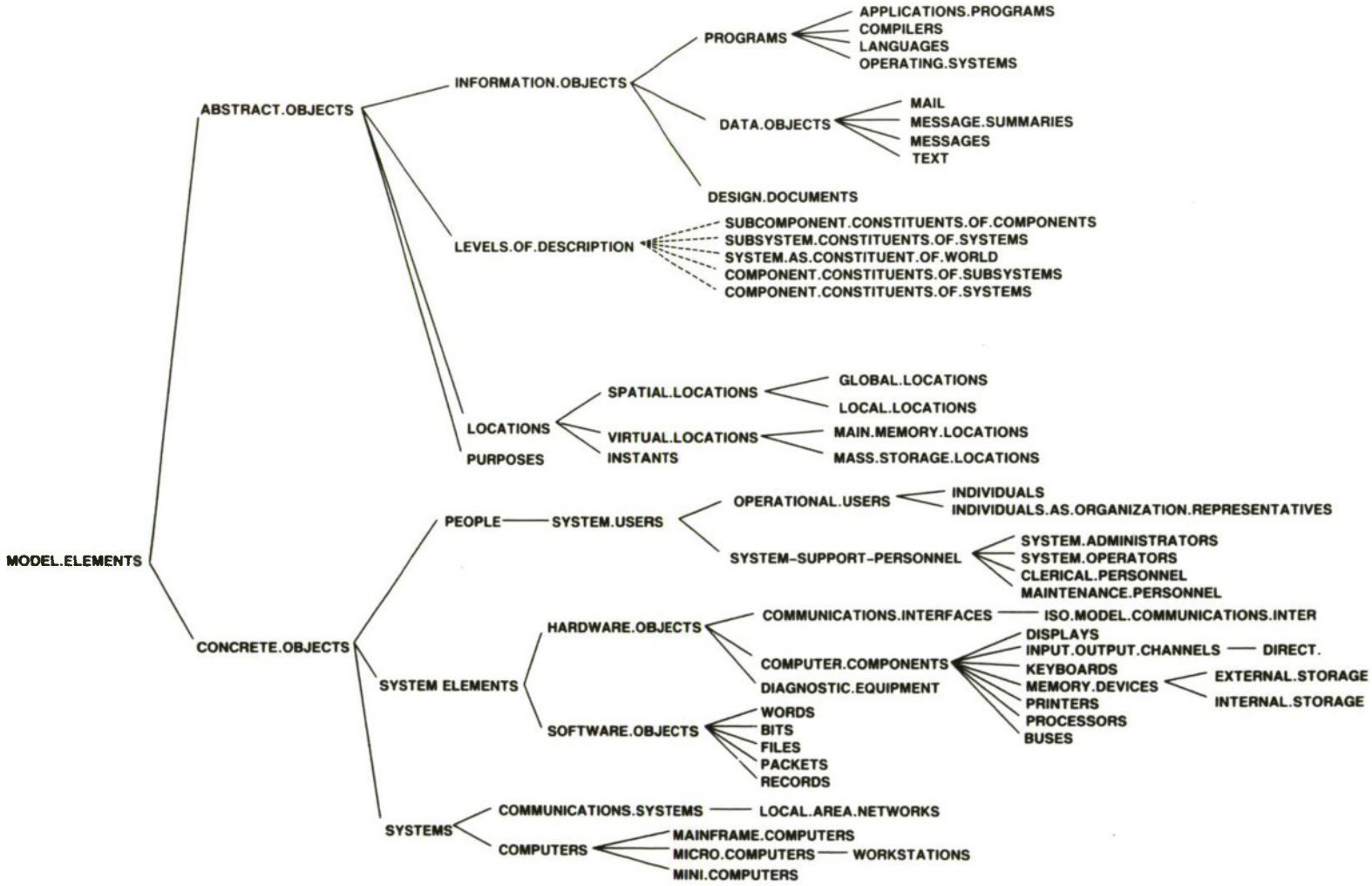

Figure 17. Model Library Structure

generated in preparing system descriptions. They must be modifiable, in the sense that changes to a user requirement or configuration allocation or design decision should be percolated through the traces, with consistency and completeness monitoring capabilities in the process.

SEIMOAR mechanisms for representing and reasoning about traceability and configuration allocation have been drawn up, but have yet to be implemented. The designs are actually quite simple. As noted earlier, KEE's frame representation language includes facets that implement procedural attachments, called active values or demons, for individual frame slots. Active values provide an ideal mechanism for automating consistency and completeness maintenance operations in the face of system model alterations. SEIMOAR will utilize active values to couple together the contents of VALUES facets for appropriate slot pairs.

For example, configuration allocation can be represented in terms of two complementary slots: a FUNCTIONALITY slot attaching to C^3I structural templates, such as hardware and software component unit classes, and a CONFIGURATION.ALLOCATION slot attaching to C^3I functional templates. Active values will ensure that additions or deletions from particular component unit FUNCTIONALITY slots will be mirrored in the appropriate functional unit CONFIGURATION.ALLOCATION slots.

Traceability can be modeled similarly. The only complication is that cross-indexing is required for slots of functional units for each combinatorial pairing of system descriptions. Function units for functional requirements, for example, will have a collection of traceability slots, one each pointing to functional units in source selection, configuration allocation, and preliminary design models. The inverse relations will be represented by complementary slots for functional units in each of those models, pointing back to functional requirements function units.

Finally, the same representational and reasoning mechanism will coordinate structural decompositions across models. The main system unit COMPONENT.STRUCTURES slot will be paired with the SYSTEM.UNIT slot pointer for component structure units in the configuration allocation. Similar relations will correlate decomposition from configuration allocation model component units to subcomponents and software modules in preliminary design description models.

4.1.3 Data and Control Flows

In order to capture preliminary design descriptions, SEIMOAR will be extended to represent data and control flows among software components and data objects. Some sort of structured analysis and design tooling framework (e.g., Yourdon, Ross, Entity-Relation-Attribute model) will be employed. SEIMOAR, it was noted earlier, is only intended to provide acquisition support through preliminary design descriptions. EASE, a companion MITRE support environment for ADA*-based systems, is being developed to handle system models from preliminary design description through actual software implementation. EASE and SEIMOAR will interface at the preliminary design description stage, through a common representation of data and control flows. This commonality will allow two-way transmission of STLDD models.

4.1.4 Reimplementation of Function Sequencing Information

SEIMOAR's present representation of function sequencing information is going to be overhauled. Currently, each function is associated with a forward rule, which indicates the conditions under which that function is to be invoked. This representation is unfortunate in several respects. First, it is difficult to determine sequencing, unless all the rules are examined or an actual behavioral simulation is run. Second, it is inappropriate to store functional relations separately from the functions themselves, since those relations contribute to the characterization of functions. The third issue pertains to the efficiency of the simulator as it is currently implemented. KEE's forward rule engine operates fairly slowly with a large rule base: indexing is only done by rule classes, not by rule clause variables. It is possible to partition the function sequencing rule base into subclasses. The problem with this strategy is that it incurs additional overhead processing costs, namely additional inferencing to identify the pertinent subclass of rules to invoke at particular stages of the simulation.

Accordingly, SEIMOAR will incorporate sequencing information into the functions themselves. Each function will contain a SUCCESSORS and a PREDECESSORS slot. The VALUES facet of these slots will be a list of lists. Each sublist will consist of the name of a possible predecessor or successor function together with a list of the conditions, formerly rule antecedents, that regulate branching. The user will only have to supply SUCCESSORS information: an active value will invert the list to fill in values for PREDECCESSORS automatically. The message formerly sent by the simulator activity

*Ada is a trademark of the Department of Defense.

method that will be applied to the SUCCESSORS list to determine proper branching. This design solves all three problems with the previous implementation noted above.

4.1.5 C^3I World Knowledge and SEIMOAR Semantics

Presently, most of the data items filled in for slot VALUES facets are simply character strings. The list of IMPLEMENTATION.REQUIREMENTS VALUES data items for the AMPE.INTERFACES unit class, for example, are simple tokens. As the library fills up with templates (e.g., dealing with communications, functions, purposes), these tokens will be transformed into pointers into the model library knowledge base. The AMPE.INTERFACES HARDWARE.REQUIREMENTS VALUES facet datum, MODEMS, refers to a model system component class with significant descriptive detail. In essence, the templates store application token semantic contents, or interpretations, in support of generalized symbolic reasoning and manipulation.

The importance of this architecture and modeling strategy should not be underestimated. Doug Lenat, at MicroElectronics and Computer Technology Corporation (MCC), is attempting to construct an encyclopedic knowledge base, intended to capture general world knowledge. His system, called CYC, is similar in principle, though obviously significantly more ambitious in scope, to SEIMOAR's modeling library. Lenat's intention is to use CYC to solve the shallowness and brittleness problems associated with current expert systems. Shallowness refers basically to heuristic knowledge, (e.g., as expressed in rules), that is not grounded by explicit domain models. Brittleness refers to the rather abrupt degradation of expert system performance at the periphery of their distinctive area of competence.

One serious problem in Lenat's strategy is that he has not yet addressed the distribution of communication and control capabilities between CYC and expert systems that it is supposed to service. In contrast, SEIMOAR is beginning to implement a specific integration strategy. Reasoning has to be directed by the application system side; otherwise CYC (or SEIMOAR) would have to know how to control reasoning in systems that employed an arbitrary number of different knowledge representation and inferencing schemes. Nevertheless, some mechanism must be instituted for retrieving information, when available, from the resource knowledge base. In SEIMOAR, this function is performed by data value tokens, treated as semantic pointers into the modeling library.

4.1.6 Static Analysis

Analyzability probably represents the greatest opportunity offered by the model-based approach to the acquisition support effort. Analysis in the acquisition support context encompasses the evaluation and comparison of design descriptions with respect to completeness, correctness, consistency, and feasibility. Analytic capabilities can be divided into static and dynamic categories. Dynamic analysis will be discussed in connection with planned enhancements to SEIMOAR's simulation capabilities.

Static analysis is defined here to refer to the verification of completeness, consistency, and technical feasibility of system structures and functions. It is intended that SEIMOAR support three kinds of static analysis: comparisons among variant system descriptions (e.g., alternative requirements or configuration allocations); comparisons of system model elements with respect to component or functional characterization standards; and perhaps most important, comparisons of system models from different stages of the program cycle (e.g., requirements vs preliminary design descriptions).

One important kind of static analysis of variant models from the same acquisition phase is sizing and costing estimation: given a set of functional requirements or requirements plus a specific high level architecture and configuration allocation, what is the expected size and cost of a system and its components?

SEIMOAR will soon incorporate a static completeness and consistency checking capability. The use of library templates for system model construction, via a copy-and-edit strategy, has already been described. Templates can also be construed as minimal, correct or "debugged" design standards. System models can be verified with respect to completeness and consistency simply by comparing system model elements and structures against corresponding library standards.

Consistency checking in this context guarantees that VALUES facet contents of structural and functional system model units conform to legal or admissible data values prescribed by library templates. Consistency checking can also apply to relational attributes: templates can prescribe legal kinds of connections between system structures, functions, and between structures and functions. Structural constraints pertain to interface requirements (e.g., that nothing can couple directly to a local area network except a LAN interface unit). Functional constraints include temporal precedence relations (e.g., functions to save or retrieve

information cannot be performed unless preceded by an appropriate information creation function). An example of the third class would be a configuration allocation proscription (e.g., that data manipulation functions must be allocated jointly to hardware and software elements).

SEIMOAR's completeness checking capability will verify that system model structural and functional elements have minimally adequate characterizations. In concrete terms, template attributes will be tagged by facets identifying them as necessary, sufficient, or optional descriptors with respect to each stage in the system development cycle. A mechanical pattern-matcher will then check model elements against the templates to ensure that attributes that are collectively necessary and sufficient to characterize system components or functions (at a given development stage) have been specified. For example, the specification of a computer might require a description of display, keyboard, processor(s), bus, main and secondary storage, printer, and power supply in order to qualify as complete.

Admittedly, completeness and consistency as defined above do by no means constitute verification in any strict, formal sense. Still the automation of even limited static analysis capabilities such as these constitute a significant step in acquisition support.

4.2 BEHAVIORAL SIMULATOR ENHANCEMENTS

The two most urgent extensions to SEIMOAR's current behavioral modeling functionality are the incorporation of a concurrent or parallel processing simulator framework and the inclusion of quantitative modeling capabilities.

4.2.1 Simulation of Concurrent Processes

The initial MCS AMH functional requirements test vehicle did not specify explicitly that system architecture might involve any parallel processing. Moreover, the description of functionality was highly sequential. Consequently, it was decided to implement initial discrete time functional simulation capability based on a sequential processing model. It can be expected, however, that C^3I systems in the future will depend increasingly on concurrent processing architectures. Heavy data loads and acceptable system response times make this technology design shift almost inevitable. It is important, therefore, that SEIMOAR's simulation capabilities be extended to cover concurrency modeling.

One of the primary objectives of SEIMOAR follow-on work is to address this modeling need. The new simulator shell will employ an object-oriented architecture, in common with the sequential process modeler. It is expected that the latter's unique system clock and activity handler will be replaced by a collection of activity handlers, each of which incorporates its own private clock. The control architecture for this arrangement (viz., synchronization, management of task assignments), has not yet been determined.

4.2.2 Quantitative (Performance) Modeling

In the first phase of the SEIMOAR project, attention was confined to schematic and qualitative modeling. Functionality was defined and subsequently modeled in terms of one or multiple units (e.g., workstations). Typically, system descriptions call out an exact number or a full capacity complement (e.g., thirty). Nothing in KEE or SEIMOAR precludes exhaustive rather than schematic system modeling.

The absence of quantitative modeling capability is a more serious limitation. The current simulator framework expends one time unit for the execution of any single function on one or more data objects. Consequently, SEIMOAR does not currently support specification or simulation-based validation of performance requirements (e.g., capacity or processing rates). Clearly, performance is a critical ingredient in technical feasibility analysis (e.g., can a proposed design meet the designated response constraints?).

Fortunately, the simulator framework of SEIMOAR can easily be adapted to accommodate quantitative modeling. Current plans are to install a TIME.EXPENDED attribute in each functional unit. The data stored here will either be a simple number or a numerical function, representing the amount of time required to accomplish function execution. It is important to be able to use a numerical function, because time expended might depend on system variables and properties of the relevant data objects. The simulator clock advance method will then be adapted to increment by a single time unit if no TIME.EXPENDED value is available, or to increment simulated time by the supplied or computed value. It will also be necessary to alter the modeling library to reflect performance constraints on functions and on system components.

4.2.3 Scenario Generator Extensions

The current scenario generator for the simulator needs to be generalized and enhanced. At present, the generator is tailored to constructing and editing a specific class of events, namely AUTODIN message objects arriving at the AMH. What is clearly needed is an event editor that is generalized to create arbitrary kinds of events. The editor is currently driven by a hand-coded list of attributes associated with AUTODIN.MESSAGES, together with a TIME.OF.ARRIVAL slot required by the simulator. The new editor will determine by itself the appropriate slots, contained in a system model data object unit class that the user is prompted to select. The editor will be guided in this task by slot facet labels inserted at the time of creation of application data object classes (e.g., mail objects).

Another enhancement to the scenario generator will be needed primarily only for large-scale simulations. The current editor creates events individually, by explicit menu prompts to the user for each object. This approach is only adequate when small numbers of events have to be generated, on the order of several dozen. For larger test scenarios, it is desirable to have a generalized bulk event generation capability, in which the user specifies distributions of event attribute values and a statistical distribution of intervals between event TIME.OF.ARRIVALS. The generator would then create a scenario by constructing a suitable event population via stochastic methods.

4.2.4 Dynamic Analysis

Dynamic analysis capabilities, grounded in system simulations, evaluate behavior and quantitative performance. SEIMOAR is intended to support two kinds of dynamic analysis. The first involves comparisons of alternate system behavioral descriptions at a given stage of the acquisition process, (e.g., requirements, source selection, preliminary designs). The second class, more difficult than the first, compares behavior across acquisition stages, in order to verify completeness and consistency of more detailed behavioral descriptions with respect to earlier ones (e.g., user functional requirements).

The basis for dynamic analysis capabilities will be behavioral simulation event logs. The simulator shell will be extended to provide a configuration option that exercises two distinct behavioral models from a given set of test scenarios. The results of each modeling run, consisting of a sequence of events, will be stored in history logs. Thus, dynamic analysis utilities will be

based on comparisons of system events in response to identical stimuli.

In this context, completeness amounts to comparing the two logs to determine whether corresponding kinds of events are present. Consistency, similarly, involves comparisons of multiple logs to determine whether corresponding kinds of events occur in corresponding sequential order.

The most difficult kind of dynamic analysis is expected to be the comparison of behavioral models across different phases of system development, such as functional requirements vs preliminary design. Such models express system functionality with different degrees of resolution. Behavior at the requirements level involves changes of state defined in the system as a whole. In contrast, behavior in a preliminary design model refers to changes of state in system components or modules. As a result, it becomes extremely difficult to map events from one behavioral simulation model onto events from another.

Current plans are to try to simplify this mapping problem by exploiting a feature of the simulator architecture: each function in a behavioral model has an explicit correspondence to a specific set of events. Users program this information explicitly in order to simulate the action of functions in the first place. The idea, then (figure 18), is to match functions across different system models and use that correspondence to map manageable subsets of events from one model onto subsets from the other.

Unfortunately, it turns out that the mapping of functions across system descriptions can be fairly messy. The acquisition standards only require that a traceability matrix be maintained throughout system development. This matrix indicates what functions in various contractor products cover the user's original functionality requirements. Generally, this mapping is not a simple one-to-one correspondence. In fact, in the worst case, it is a many-to-many mapping (figure 19). It is hoped that the literature on program verification and mathematical fields such as topology and set theory will provide some formal assistance in solving this problem.

4.3 USER INTERFACE ISSUES

The current user interface for SEIMOAR consists of KEE user interface capabilities for system model (knowledge base) creation, modification, and browsing, and a menu-driver with special graphic

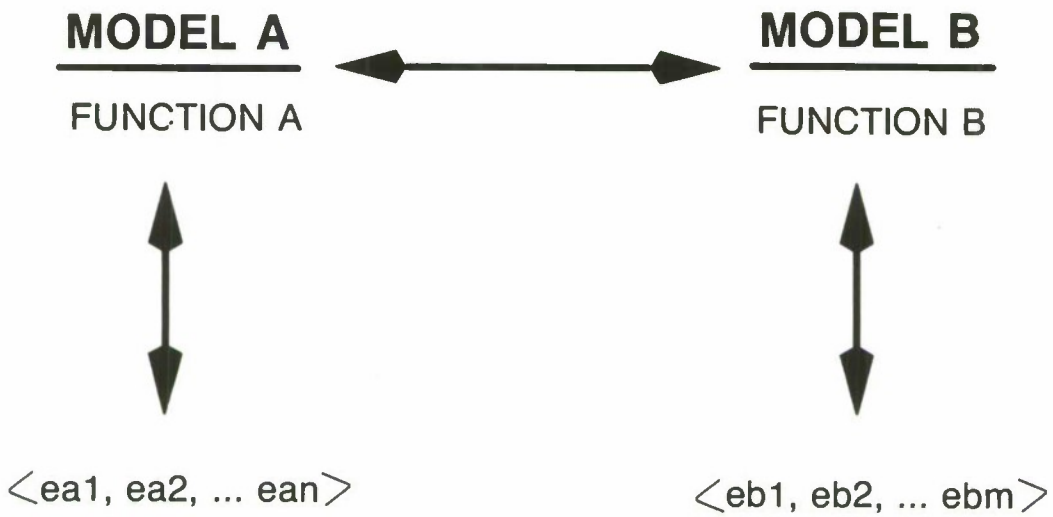

Figure 18. Dynamical Analysis

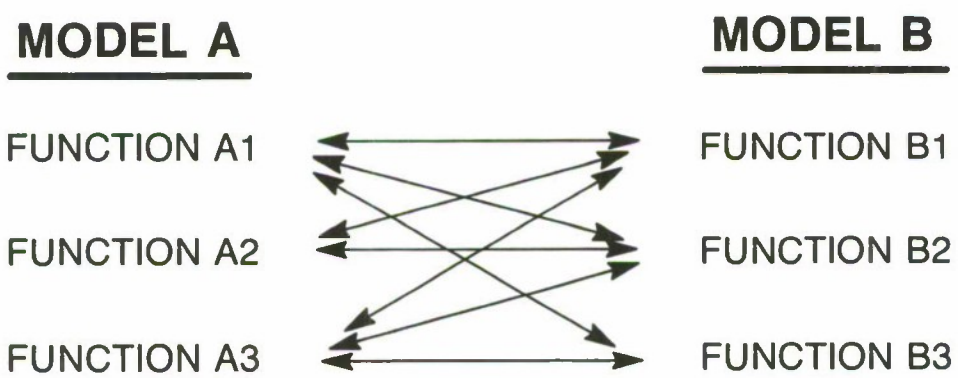

Figure 19. Function Traceability

displays and traces for the simulator. A full-fledged interface, customized for a general acquisition support user community is being planned. It is assumed that the users are expert in system design and analysis but not in artificial intelligence representations and programming techniques.

Experimentation with icons in the first version of SEIMOAR has resulted in an improved understanding of appropriate implementation and use of icons in the interface. The initial version of SEIMOAR maintains icons as distinct units in a knowledge base separate from the main system model. Utility methods are used to link icons to model components. The connections between graphic icons bear no relation to connections between component units that they represent. Icons can be connected, via mouse-activated methods, to icons representing substructures of the moused icons. No capability to access the component data structure by mousing the linked icon was provided.

4.3.1 Incorporating Icons into Library Templates

In follow-on effort, icons will be incorporated directly into model library templates, not only for system components but for functions as well. Multiple inheritance will be employed to mix a specialized image class into functional and structural library templates. Thus, each template will have two sets of attributes, those characterizing a C^3I function, system, or system component object class and those characterizing graphic objects that correspond to those classes.

4.3.2 Structure and Function Editors

Icons will be used to drive a capability to browse system models. More important, graphic relations among icons will correspond directly to relationships among associated components and functions: model relationships (e.g., allocation, architectural decomposition, traceability, function sequencing), will all be specifiable via iconic structure and function editors.

This list sounds deceptively ambitious. In fact, the underlying implementation mechanisms are few in number and simple in design. Basically, the image class mixin discussed in 4.3.1 will incorporate mouse button functions. For example, buttoning the mouse on a structural icon will invoke a menu offering architectural decomposition, allocation, or data/control flow editing options. Actions will then be performed on the appropriate component unit (or unit class) slots. Similar options for allocation, traceability,

sequencing, and functional flow editing activities will be provided when users mouse on functional icons.

Thus a single set of editor functions will accommodate several model generation activities. The mouse buttoning context (i.e., the kind of template moused or menu option selected), determines the functionality and slots to be modified by the generalized editor mechanisms. It is also simple to guarantee that the editors permit only semantically correct operations. For example, the structural decomposition editor can be constructed to guarantee that system components are only connectible in legal (i.e., domain-sensible), configurations. The real drivers here are library structural template slots, which will be designed with facets (e.g., OK-CONNECTIVITY), that enumerate admissible connections between structural classes. The editor has only to check the moused data entry against these value constraints. Similar semantic capabilities can be designed into the other structural and functional editors mentioned above.

4.3.3 Declarative Specification of Behavior and Animation

Another interface requirement for a general acquisition user community is that function behaviors be declaratively specifiable. SEIMOAR currently requires explicit programming to characterize the state changes in model objects which represent function actions and the state changes required for underlying simulator mechanics. The new interface will solicit user menu selections. The menu options will identify the relevant objects to be changed, the slots to be changed, and the altered values. SEIMOAR will convert these menu choice entries into appropriate LISP and KEE utility function calls to change the system, data, and simulator model object states.

A similar menu-based interface will facilitate programming of iconic animations for functional behaviors. It is not clear at this time how much of the animation can be programmed directly from the behavioral description and how much users will have to declare explicitly.

4.3.4 Automated System Model Initialization

Another interface requirement is to automate some of the initial mechanics of system model setup. The setup capability will create a new knowledge base, and copy in an overall skeleton of units, with appropriate methods and attributes inherited from library templates, to start building models. The units will include the graphic image mixin described above. In addition, the

initialization function will copy over the behavioral simulator and SEIMOAR user interface.

In addition, SEIMOAR will incorporate a template copier function that automatically copies a specified template, and establishes a pointer to the source template for subsequent static analysis utilities. This copy function, specialized from KEE to SEIMOAR, automatically creates the pointers required to drive the planned static analysis capabilities. Additional variants of KEE utility functions, adapted to purposes specific to SEIMOAR, will further customize the user interface to the tool.

4.3.5 Model-driven Document Generation

It turns out, not too surprisingly, that the SEIMOAR models of C^3I system requirements and contractor system descriptions do not correspond very well to the DOD specification format. DOD documentation disperses references to system elements and features across a variety of contexts (paragraphs). The KEE frame-based representation, in contrast, emphasizes compactness and aggregation of references to a constrained number of data structures. Nevertheless, the DOD standard is extremely uniform in structure, as are SEIMOAR's system models. This suggests that a mapping or translation capability is feasible for converting system models into military standard acceptable views, either within KEE or in the form of explicit documentation.

The grounds for this capability, as always, would be identifying tags (i.e., 2167 paragraph numbers for each topic in figure 1), inserted at unit and slot levels, via a KEE method that drives a label selection menu. Appropriate categories of information can be collected using these tags (e.g., paragraph 3.3 of SSS) and displayed, printed, or stored in ASCII (and graphics) files. The mediating interface is an English(like) front-end parser, that converts KEE frame data into complete English sentences complete with "shalls."

The document generation capability will not be added to SEIMOAR for some time, in view of the implementation requirements for more urgent features and functions noted above. Nevertheless, it is important to identify this particular avenue of tool evolution, and the availability of current "natural language" processing AI technology to implement such a capability.

SECTION 5

SUMMARY

The initial version of SEIMOAR constitutes a proof-of-concept investigation into a knowledge-based approach to acquisition support (or in-house system development). SEIMOAR is intended to facilitate the construction and analysis of models representing functional requirements and contractor system development products through preliminary design description. Initial tooling efforts resulted in a modeling library, containing generic templates for C^3I system components and functions, and a sequential processing discrete time functional behavior simulation shell. SEIMOAR was designed through analysis of functional requirements for a test vehicle C^3I system, an automated message-handling system. SEIMOAR successfully supported creation of structural, functional, and behavioral models of the test vehicle.

"Knowledge" is distributed across a variety of structures in SEIMOAR: the underlying development environment and the basic knowledge representation superstructure (frames, demon and method procedural attachments); C^3I expertise encoded in modeling library templates; specification and design information in system model knowledge bases; the discrete event simulator; and the planned pattern-driven analysis utilities.

SEIMOAR was implemented on top of KEE, a commercial AI hybrid tool shell. The primary representational elements include frames, rules, object-oriented procedural attachments, and graphic icons. KEE's interface programming features are used in SEIMOAR to facilitate easy, visually-oriented exploration of complex system model structures, functions, and behaviors.

BIBLIOGRAPHY

Aikins, J., "Prototypical Knowledge for Expert Systems," Artificial Intelligence, Vol. 20, No. 2, pp. 163-210, 1983.

Bobrow, D., (ed), "Qualitative Reasoning About Physical Systems," MIT Press, Cambridge, MA, 1985.

Borgida, A., Greenspan, S., and Mylopoulos, J., "Knowledge Representation as the Basis for Requirements Specifications," IEEE Computer, Vol. 18, No. 4, pp. 82-91, 1985.

Fikes, R., and Kehler, T., "The Role of Frame-Based Representation in Reasoning," Communications of the ACM, Vol. 28, No. 9, pp. 904-920, 1985.

Friedland, P., and Kedes, L., "Discovering the Secrets of DNA," IEEE Computer, Vol. 18, No. 11, pp. 49-69, 1985.

Hayes-Roth, F., Waterman, D., and Lenat, D., (eds), Building Expert Systems, Reading, MA: Addison-Wesley, 1983.

Hollan, J., Hutchins, E., and Weitzman, L., "STEAMER: An Interactive Inspectable Simulation-Based Training System," AI Magazine, Vol. 5, No. 2, pp. 15-27, 1984.

Kunz, J., Kehler, T., and Williams, M., "Applications Development Using a Hybrid AI Development System," AI Magazine, Vol. 5, No. 3, pp. 41-54, 1984.

Lenat, D., Prakash, M., and Shepherd, M., "CYC: Using Common Sense Knowledge to Overcome Brittleness and Knowledge Acquisition Bottlenecks," AI Magazine, Vol. 6, No. 4, pp. 65-85, 1986.

Sowa, J., Conceptual Structures: Information Processing in Mind and Machine, Reading, MA: Addison-Wesley, 1984.

Stefik, M., Bobrow, D., Mittal, S., and Conway, L., "Knowledge Programming in LOOPS: Report on an Experimental Course," AI Magazine, Vol. 4, No. 3, pp. 3-12, 1984.

Stefik M., and Bobrow, D., "Object-Oriented Programming: Themes and Variations," AI Magazine, Vol. 6, No. 4, pp. 40-62, 1986.

APPENDIX A

KNOWLEDGE REPRESENTATION MODELS EXPLAINED

This appendix provides a brief overview of two of the most popular kinds of artificial intelligence representational frameworks, rules and frames. This information is intended to help novices understand terminology and basic concepts referenced in the body of this report. This review is quite skeletal and abbreviated: for more leisurely, comprehensive, and skillful introductions, the interested reader is urged to study the introductory texts and articles that are available in ever-increasing numbers.

The original AI representation framework for expert systems relies on data structures called production rules. Briefly, rules encode situation-action knowledge in the form of conditional rules. Rule antecedents, or "if clauses," encode triggering patterns. Examples might include medical or equipment test results, signal patterns, or observable phenomena such as overt patient appearance or weather conditions. Rule consequents, or "then clauses," encode responses to triggering patterns. Actions might be inferences, such as diagnostic deductions based on symptom patterns, or specific behaviors, such as the prescription of medication, the issuance of particular repair, configuration, or control instructions.

Rule-based systems generally rely on one or both of two kinds of reasoning models. In forward chaining inferences, reasoning proceeds from premises to conclusions (i.e., given the presence of the former in the knowledge base, the conclusions are asserted). In backward chaining, the conclusion is posited as a hypothetical or goal, and the engine then attempts to establish the premises, which may have to be taken as intermediate goals, and so on. The goal is asserted into the knowledge base only if the premises are all verified.

A more recent and increasingly popular kind of representational model is based on the notion of frames. Frames are data structures that represent classes of objects (e.g., message processing functions, computers), or individual members or instances of classes (e.g., message collation function, workstation123). Frames aggregate together information describing individuals or classes in substructures called slots. Slots represent attributes or relations, and are somewhat analogous to field names in a conventional data base dictionary.

A frame representing the class people, for example, might contain slots encoding properties, such as HEIGHT, AGE, RACE, GREATEST-AGE WORLD-POPULATION, predicates (true-false properties) such as ALIVE?, and relations, such as SIBLINGS, FATHER, MOTHER. Loosely speaking, properties and predicates characterize individuals or classes, while relations represent links or pointers between individuals and classes. Note that HEIGHT, AGE, RACE, and ALIVE? refer to properties of persons, or individual members of the class PEOPLE, while GREATEST-AGE and WORLD-POPULATION refer to attributes of a class as a whole (the oldest member and the total number of members).

Frame slots in turn have descriptors, which are called facets. The most important facet is called the VALUES facet, which contains data items representing particular values of slots. For example, the VALUES facet for the slot COLOR for the class frame COAL might contain the datum BLACK. Thus, VALUES facets are analogous to data fields in individual records.

Slots are often equipped with facets other than VALUES. One important facet, called OK-VALUES or VALUECLASSES, characterizes legal data or data classes for slot VALUES facets (e.g., numbers vs symbols, names of people, (1 or 2 or 3)). This facet is useful for ensuring qualitative correctness of information entered into a knowledge base (e.g., appropriate value range and uniqueness of employee salaries). OK-VALUES facets encode semantic constraints on slot values symbolic entries.

One of the most important characteristics of frame-based representations is the notion of inheritance. The ability to define class, subclass, and individual frames, with appropriate relations between these entities is known as abstraction. Abstraction makes it possible to associate information (or information types) common to many individuals or classes with a more generic class. This results in substantial data compression -- information that would otherwise be repeated extensively is coded once, at an appropriate level of commonality (class abstraction). For example, all kinds of fruit, and all instances of those kinds, have certain properties (e.g., color, size, weight), in common.

Frames-based (and object-oriented) representational models support abstraction using two mechanisms, class relations and inheritance. Class relations, notably class-subclass and class-member, link two frames together. Successive class-subclass relations among a set of frames establishes a class hierarchy. The standard biological taxonomy system is a good example.

Class relations drive inheritance. Inheritance here basically amounts to information sharing. Information can be shared either between a frame representing a general class and others representing specialized subclasses, or between a class frame describing prototypical members and frames representing particular individuals belonging to the class. In virtue of inheritance relations, any information ascribed to the general class mammals holds true of any subclasses, such as cows or whales.

Class attributes and relations often have nominal or expected values (e.g., the typical VALUES of COLOR of GRANNY.SMITH.APPLES is the datum GREEN). Inheritance ensures that assignments of VALUES data to class attributes are assumed as defaults by individual members (or subclasses). Typically, if an individual deviates from the prototypical "default" or nominal characteristics in one or more respects, those individual differences (i.e., attribute values) replace or override" the inherited attributes.

Frames are very good at representing static structures and relations between objects or classes, but poor at representing procedural information (e.g., programs). Two kinds of procedural attachments are used to rectify this shortcoming, active values and methods.

Active values are pieces of LISP code contained in special slot facets, which are invoked, as relevant, when items are added, deleted, or replaced in the slot's values facet. Active values are useful in safeguarding knowledge base consistency (e.g., guaranteeing appropriate adjustments to departmental budgets whenever member employee salaries are modified or staff sizes adjusted).

Active values, also called demons, reflect slot-level procedural information. It is also important to represent frame-level procedures, which pertain to operations on, or communications between frames. To meet this need, some frame systems incorporate methods, frame-level procedural attachments borrowed from object-oriented knowledge models. Briefly, pieces of code are attached to frame class or instance slots. This code is activated by sending a message to the desired frame, consisting of the name of the method and any necessary arguments. Nothing needs to be known about the internal structure of the method itself except its name and the arguments it needs. This characteristic is known as encapsulation.

Objects are very similar to frames except that they lack facets. Object-based systems support classes, class instances, attributes (class and instance variables), and inheritance of attribute values. Given that active values are normally implemented

via facets, unless some programming tricks are played with methods (known as "wrappers"), object-based systems typically do not support slot-level procedural attachment.

Frames and rules can be combined into hybrid systems. The most popular approach is to embed frames in rules. That is, rule clauses make reference to frames and frame slot values in much the same way as conventional programming applications retrieve and add information to data base records. Typically, rule antecedents pattern match on frame slot values, comparing or testing retrieved values against one another or test standards. Rule consequents, similarly, modify the contents of frame slot values, reflecting diagnostic inferences or control actions on system models.

Alternatively, rules can be embedded in frames. In this model, frames are arranged in a hierarchy, reflecting a flow of activities or control. Within a given frame, a restricted subset of rules can be activated, to drive rule-based reasoning. In this model, frames partition rules into indexed subsets, corresponding to the relevant rules to reason with respect to specific activities. This model has not been used very often, but it has some interesting possibilities.

APPENDIX B

KEE AND ART -- COMPARISONS AND LESSONS LEARNED

One of the purposes of the SEIMOAR project was to evaluate the use of the best available commercial AI system building shells. KEE (Intellicorp) was selected because of economics. (MITRE already owned a license and a copy of the program and subsequent copies are discounted substantially.) The author has also had experience with KEE's leading competitor, a tool called ART. This appendix provides a summary comparative analysis of the two tools, together with an overall personal assessment of the value of commercial shells.

KEE is an advanced AI programming shell currently running on LISP minicomputers (LMI, Symbolics, Xerox, TI). KEE combines a powerful high-level representation language with multiple control strategies (e.g., rules, active values, methods, VALUECLASS facets), for managing reasoning about encoded knowledge. User-supplied LISP code is readily embeddable in KEE data structures. The program provides a variety of tools for building sophisticated user interfaces, most notably graphic icon libraries and editors, menus, windows, and mouse pointers. Finally, KEE integrates the above features with a powerful development environment: language editors, browsers for both application and KEE (system) structures, and program monitors and debuggers. In short, KEE provides a generic but flexible support environment for rapid prototyping of expert systems and other AI applications.

KEE basics are easy to learn and use, whether or not one is intimately acquainted with artificial intelligence. Theoretical and practical familiarity with the tool are facilitated through lecture notes and well-documented walkthrough tutorials involving small training knowledge bases.

Some caveats accompany the use of a hybrid tool such as KEE. First, nontrivial applications of KEE require a basic competence in writing LISP code (e.g., for designing methods and demons). Second, the backward-chaining rule system and rule-based inference monitoring aids, while quite powerful, are complex. Familiarity with goal-driven reasoning and some significant effort are needed to master these aspects of KEE. The rule trace and debugging capabilities are powerful, but also quite numerous and often bewildering. Third, KEE (Version 2.1), lacks important representation and reasoning mechanisms that are available in a competitor hybrid AI tool called ART (Automated Reasoning Tool, Inference Corp., Los Angeles, CA).

ART's architecture rests on a base declarative language derived from the predicate calculus, which incorporates a frame-like construct known as a schema. Schemata have slots, which in turn take on values. Schemata can be connected through standard set-theoretical relations (class-subclass, class-member), supporting information inheritance. However, schemata lack facets and procedural attachments of any kind. Consequently, ART's data structures are passive, while KEE's frames, in virtue of demons and methods, are active. ART's primary reasoning mechanisms are backward and forward chaining rule inference engines. ART also provides graphics and language editors, program monitors, and so forth, although none of these features are as powerful as KEE's.

ART's strongest feature is a representational model known as viewpoints or possible worlds. Possible worlds support an inference mechanism called hypothetical reasoning. In effect, possible worlds drive the capability to generate multiple, alternative "what-if" models in parallel, allowing comparisons between the different hypothetical simulations. ART's rule-based approach to hypothetical reasoning provides a natural vehicle for activities such as planning, scheduling, and forecasting. These complex tasks typically involve large numbers of interacting variables and knowledge that takes the form of constraints (e.g., any job shop schedule at plant Z that calls for more than 1000 manhours of labor per week is unimplementable, and should be discarded).

Possible worlds also ground truth-maintenance systems (TMS), a popular approach to the problem of nonmonotonicity. In monotonic models, information simply accumulates. For example, many kinds of static classification tasks take input evidence and incrementally refine identification to increasingly more specific categories. No assertions or intermediate inferences are ever falsified or retracted. In contrast, nonmonotonic models are dynamic, involving information that changes over time. For example, in a battlefield management scenario, the movements of friendly and enemy forces often result in situations wherein reasoning and decisions based on earlier field positions and trends might become obsolete or otherwise invalidated. A TMS enables a decision support system to trace and retract all such unreliable inferences and conclusions, irrespective of the complexity of the prior chain of reasoning. It is extremely difficult to deal with hypotheticals and nonmonotonicity using KEE, which lacks possible worlds structures.

New versions of KEE and ART have been released recently. KEE Version 3.0 addresses most of the representational and reasoning shortcomings of Version 2.0 (TMS, possible worlds, user-definable relations, hypothetical and temporal reasoning, improved backward-chaining). However, the complexity of KEE's

representational architecture has increased correspondingly. The new capabilities are not as easily integrable within KEE's frame-based system as they were in ART's underlying propositional representation. Moreover, just as KEE has improved its functional capabilities, ART appears to have improved its capabilities (demon and method procedural attachments), its efficiency, and its user interface.

Licensing and training costs for ART and KEE are expensive. Moreover, ART and KEE, like all tools, often constrain and inhibit implementation of design, as well as facilitate application development. Weighed against these costs, however, are considerable advantages: powerful, prefabricated development environments, externally supplied documentation, training, consultation, enhancements, and product control.

The costs of not using commercial tooling must also be appreciated: loss of in-house tooling expertise due to designer staff turnover, overhead and uncertain availability for customized tooling and support, nonexistence of documentation, brittleness of resulting systems to nondesigners, low code reusability, and so forth. Customized tooling tends to be no less constraining and inhibiting upon occasion as commercial systems. Weighed against the access to source code is the absence of documentation and product assurance methods and standards.

My personal verdict at this point is this: both tools are eminently suitable for rapid prototyping and proof-of-concept experimentation. I have no experience using the tools for full-scale production systems. Consequently, it is not clear to me that commercial tools will support application scaling and system performance requirements.

This does not imply that commercial shells should be abandoned when embarking upon development projects for large scale or performance critical systems. Rather, it means that rapid prototyping up through early design must be followed by a formal assessment of the adequacy of the commercial tool for production system requirements as opposed to an optimized, custom built version of the system. In either case, the commercial tool will have served its purpose as an early development, top-down design vehicle.

(Amortized over multiple projects, a commercial shell is more cost-effective than internally developed project-dedicated tooling. By the implementation phase, system development calls for bottom-up rather than top-down methodology. At this point, single pass custom tooling is more likely to involve acceptable time and cost. Using

customized tooling throughout the development cycle, which generally involves one or more major redesigns, is a much more expensive proposition.)